Weakly Sensitive Analysis for JavaScript Object-Manipulating Programs

Yoonseok Ko, Xavier Rival, Sukyoung Ryu

Weakly Sensitive Analysis for JavaScript Object-Manipulating Programs[†]

Yoonseok Ko[1*], Xavier Rival[2] and Sukyoung Ryu[1]

[1] *Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea*
[2] *DIENS, École Normale Supérieure, CNRS, PSL Research University and INRIA*

SUMMARY

While JavaScript programs have become pervasive in web applications, they remain hard to reason about. In this context, most static analyses for JavaScript programs require precise call-graph information, since the presence of large numbers of spurious callees significantly deteriorate precision. One of the most challenging JavaScript features that complicate the inference of precise static call graph information is read / write accesses to object fields the names of which are computed at runtime. JavaScript framework libraries often exploit this facility to build objects from other objects, as a way to simulate sophisticated high-level programming constructions. Such code patterns are difficult to analyze precisely, due to weak updates and limitations of unrolling techniques. In this paper, we observe that precise field correspondence relations can be inferred by locally reasoning about object copies, both regarding to the object and to the program structure, and we propose an abstraction which allows to separately reason about field read / write access patterns working on different fields, and to carefully handle the sets of JavaScript object fields. We formalize and implement an analysis based on this technique. We evaluate the performance and precision of the analysis on the computation of call-graph information for examples from jQuery tutorials.

Received . . .

KEY WORDS: JavaScript; static analysis; trace partitioning abstraction; object abstraction; string abstraction; abstract interpretation

1. INTRODUCTION

In recent years, JavaScript has been increasingly used for client-side and server-side web applications due to its easy operation in development and deployment with scripting language characteristics such as dynamic and interpreted language features. On the other hand, the drawbacks induced by these characteristics have led to various attempts to statically compute semantic properties of JavaScript programs to support tools for type checking [1, 2], optimization [3], and verification of the absence of private information leakage [4].

Meanwhile, a precise static call graph construction has become an important issue in static analysis of JavaScript programs. Many works have revealed that a naïve approach causes a large number of spurious callees [5, 6, 7, 8]. Since analysis of semantic properties such as the absence of private information leakage requires inter-procedural information, the large number of spurious callees significantly degrades the scalability and precision of the analysis.

*Correspondence to: Yoonseok Ko, School of Computing, Korea Advanced Institute of Science and Technology, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea. E-mail: mir597@kaist.ac.kr
[†]The results in this paper are based in part on findings presented in the proceeding of APLAS'17.

```
1  function fix( e ) {
2    i = copy.length;
3    while ( i-- ) {
4      p = copy[ i ];
5      t = oE[ p ];
6      e[ p ] = t;
7    }  // oE.x ⋈ e.x, oE.y ⋈ e.y
8  }
9  var oE = {x: f1, y: f2}, o = {};
10 fix( o );
11 o.x();
```

Figure 1. A simplified excerpt from jQuery 1.7.2 and an example use case

One of the problematic JavaScript features that complicate the precise static call graph construction is read / write accesses to object fields the names of which are computed at runtime. In JavaScript, an object has a set of fields, each of which consists of a name and value pair, and the field lookup and update operations take the value of an expression as an index that designates the field to read or write. JavaScript framework libraries often exploit this facility to build an object from another object by copying all the fields one by one (shallow copy) or by computing values using the fields in the source object (e.g., to simulate accessor methods). To reason over such *field copy or transformation* patterns (for short, FCT patterns), analysis tools need to resolve precisely the fields of objects, and the relations among them.

As an example, we consider the jQuery implementation of class-inheritance using field copies of an object as shown in Figure 1. To achieve that, function `fix` copies the fields of `oE` designated as the elements of array `copy` into array `e`. This copy of the fields of object `oE` takes place in the loop at lines 3–7. More generally, an FCT pattern boils down to an assignment of the form `o1[`f`(v)] =` g`(o2[v])`, where `v` is a variable, `o1` and `o2` are two objects, f is a function over field names and g is a function over values. We call variable `v` the *key variable* of the FCT pattern. In Figure 1, the statements at lines 5 and 6 define an FCT pattern `e[p] = oE[p]` with the key variable `p`, where f and g both are the identity function.

When fields get copied store closures, static resolution of function calls requires precise field relation information. The call at line 11 in Figure 1 is such a case. Therefore, computation of a call-graph for this code requires a precise analysis of the effect of the FCT pattern at lines 5 and 6. Indeed, to determine the function called at line 11, we need to observe that the value of the field `x` in object `e` is a copy of the value of the field `x` in object `oE`. We write `oE.x` $\bowtie$ `e.x` for this *field correspondence relation*. A basic points-to analysis that applies no sensitivity technique on the loop (such as loop unrolling) will fail to precisely infer this field correspondence relation, hence will compute spurious call edges in a call-graph. Indeed, it would consider that at line 11, `o.x()` may call not only `f1` (that is actually called when executing the program) but also the spurious callee `f2` in the call-graph.

In this paper, we design a static analysis for the inference of precise field correspondence relations in programs that use FCT patterns so as to compute precise call graph information. The inference of precise field correspondence relations is especially difficult when it takes place in loops (as in Figure 1) due to the possibly unbounded number of replicated fields. It significantly complicates the analysis of meta-programming framework libraries like jQuery. To achieve the inference of precise field correspondence relation, we observe that, while some loops can be analyzed using existing sensitivity techniques, others require a more sophisticated abstraction of the structure of objects, which can describe field correspondence relations over unbounded collections of fields. The goal of this analysis is to statically compute the precise call-graph information, and it also applies to other analyses that require tracking of values propagated in FCT patterns.

Contributions. The contributions of this paper are as follows:

DOI: 10.1002/spe

- We identify the field copy or transformation patterns into two categories according to their inference of field correspondence relations, and characterize existing techniques into the categories (Section 2).
- We formally define the semantics of field correspondence relations that clearly explains the phenomenon causing a large number of spurious callees (Section 4).
- We present new abstractions to represent program executions with precise field correspondence relations (Section 5).
- We construct a new analysis that infers precise field correspondence relations for programs that perform field copies and transformations (Section 6).
- We evaluate the analysis with micro-benchmarks and real code (Section 8).

2. CATEGORIES OF FIELD COPY OR TRANSFORMATION (FCT) PATTERNS

In this section, we study FCT patterns, and draw the line between two categories of such patterns. We define the categories according to how existing static analyses for JavaScript programs cope with them. More precisely, we consider how analyses reason over FCT patterns, and infer field correspondence relations for each FCT pattern. Based on this discussion, we show the need for new semantic abstraction techniques to analyze such cases that the existing techniques cannot handle with a sufficient level of precision.

Class of static analyses of interest In this paper, we focus on static analyses for JavaScript programs that aim at computing reachability information, so as to verify safety properties for example. These include state-of-the-art JavaScript analyzers such as JSAI [9], SAFE [6], and TAJS [5].

These analyses compute an *over-approximation* of the *reachable states* of a program in a *forward manner*. More specifically, they compute an over-approximation of the set of memory states that can be observed in program executions. We assume that a memory state is made of an environment which maps variables to their values, and of a set of *objects* that are sets of field name and value pairs. We assume that strings designating field names, locations designating objects, and function closures are considered as values in programs. Furthermore, we consider analyses that feature abstract field names, abstract locations, and abstract values that represent concrete field names, locations, and values, respectively. Since the number of fields a JavaScript object may contain during an execution is unbounded, such analyses also need to provide a notion of *abstract field* to represent a *set* of pairs made of a field name and its value.

FCT patterns While we let an FCT pattern be a series of statements that starts with a read access and ends with a write access to object fields the names of which are computed at runtime in Section 1, we use the following definition from now on:

Definition 2.1. An FCT pattern consists of an assignment of the form:

$$\texttt{o1}[f(\texttt{v})] \;=\; g(\texttt{o2}[\texttt{v}]),$$

where `v` is a variable, `o1` and `o2` are two objects, f is a function over field names, and g is a function over values.

We abstract the computation of the field name of `o1` using `v` as f and the computation of the assigned value using `o2[v]` as g. We call the fields of object `o2` that are read during the computation of an FCT pattern the *fields involved* in the FCT pattern.

Example 2.1. Figure 2 shows a loop that contains an FCT pattern at lines 4–6 where variables `v`, `o1`, and `o2` respectively correspond to the same named variables in the FCT pattern, $f(\texttt{v})$ on the

DOI: 10.1002/spe

```
1   var i = arr.length, k, t, v;
2   while ( i-- ) {
3     v = arr[i];
4     t = o2[v];
5     k = v.toUpperCase();
6     o1[k] = t;
7   }
```

Figure 2. An example loop with an FCT pattern

left-hand side corresponds to `v.toUpperCase()`, and g on the right-hand side corresponds to the identity function.

The loop iterates over the elements of array `arr`, and copies the fields of object `o2` to object `o1`. Each iteration copies the field designated by an element of array `arr` from object `o2` to object `o1` with capitalized field name computed by the function `toUpperCase`. Since multiple fields are copied to object `o1` with capitalized field names during the loop execution, precise field relations are important to identify each field name and value pair in object `o1`.

All the fields in object `o2` designated by the elements of array `arr` are involved in this FCT pattern.

Categories of FCT patterns We now define two categories of FCT patterns based on the number of concrete fields involved in the patterns: the first category corresponds to the case where a single field is involved, and the second to the case where several fields may be involved.

To explicitly denote the number of fields involved in an FCT pattern for a given pre-condition $\mathbb{S}$, we use an auxiliary function $\mathsf{readfields}_{\mathbb{S}}$ indexed by $\mathbb{S}$, which takes two variables as arguments and returns the set of the fields designated by the value of the second argument in an object designated by the first argument. For example, $\mathsf{readfields}_{\mathbb{S}}(\texttt{o2}, \texttt{v})$ represents the set of all the fields in object `o2` designated by the value of variable `v` in the pre-condition $\mathbb{S}$. In an FCT pattern $\texttt{o1[}f(\texttt{v})\texttt{]} = g(\texttt{o2[v]})$ with a concrete pre-condition $\mathbb{S}$, $\mathsf{readfields}_{\mathbb{S}}(\texttt{o2}, \texttt{v})$ represents the set of all the fields involved in the FCT pattern because they are the fields read during the computation of the FCT pattern.

When only one field is involved in an FCT pattern, we call the FCT pattern *singular*.

Definition 2.2 (Singular FCT Pattern)**.** An FCT pattern $\texttt{o1[}f(\texttt{v})\texttt{]} = g(\texttt{o2[v]})$ is categorized as a singular FCT pattern with respect to pre-condition $\mathbb{S}$ when only one field is read during the evaluation of the expression `o2[v]` in $\mathbb{S}$:

$$|\mathsf{readfields}_{\mathbb{S}}(\texttt{o2}, \texttt{v})| = 1.$$

Example 2.2 (Singular FCT Pattern)**.** We consider the FCT pattern of the code fragment shown in Figure 2 and assume a concrete pre-condition in which a single concrete field is involved in the pattern. To illustrate such a case, we assume that in the concrete pre-condition of the FCT pattern at lines 4–6, object `o2` has two fields named `x` and `y`, respectively storing closures `fx` and `fy` as values, and that the value of the key variable `v` is `x`. Since only one field named `x` is read, a single field is involved in the pattern.

Figure 3 shows an analysis of this singular FCT pattern. The *concrete post-condition* represents all the execution results that can be observed when starting from the given concrete pre-condition. The concrete post-condition shows that the field `x` of object `o2` is copied to object `o1` with a capitalized field name `X` computed by the function `toUpperCase`. We give both concrete and abstract pre- and post-conditions. The abstract post-condition shows that the analysis precisely computes the value of the field that is copied in object `o1`.

When the set of fields involved in an FCT pattern contains strictly more than one element, we call the FCT pattern *plural*.

DOI: 10.1002/spe

Concrete Pre-condition	Abstract Pre-condition
o2 $\mapsto$ {x : fx, y : fy}	o2 $\mapsto$ {x : fx, y : fy}
v $\mapsto$ x	v $\mapsto$ x

```
t = o2[v];
k = v.toUpperCase();
o1[k] = t;
```

Concrete Post-condition	Abstract Post-condition
o1 $\mapsto$ {X : fx}	o1 $\mapsto$ {X : fx}
o2 $\mapsto$ {x : fx, y : fy}	o2 $\mapsto$ {x : fx, y : fy}
v $\mapsto$ x	v $\mapsto$ x
k $\mapsto$ X	k $\mapsto$ X
t $\mapsto$ fx	t $\mapsto$ fx

Figure 3. Example analysis of a singular FCT pattern

Concrete Pre-condition	Abstract Pre-condition
o2 $\mapsto$ {x : fx, y : fy}	o2 $\mapsto$ {x : fx, y : fy}
v $\mapsto$ x $\vee$ y	v $\mapsto$ x $\vee$ y

```
t = o2[v];
k = v.toUpperCase();
o1[k] = t;
```

Concrete Post-condition	Abstract Post-condition
o1 $\mapsto$ {X : fx, Y : fy}	o1 $\mapsto$ {X : fx $\vee$ fy, Y : fx $\vee$ fy}
o2 $\mapsto$ {x : fx, y : fy}	o2 $\mapsto$ {x : fx, y : fy}
v $\mapsto$ x $\vee$ y	v $\mapsto$ x $\vee$ y
k $\mapsto$ X $\vee$ Y	k $\mapsto$ X $\vee$ Y
t $\mapsto$ fx $\vee$ fy	t $\mapsto$ fx $\vee$ fy

Figure 4. Example analysis of a plural FCT pattern

Definition 2.3 (Plural FCT Pattern)**.** An FCT pattern `o1[`f`(v)] = `g`(o2[v])` is categorized as a plural FCT pattern with respect to pre-condition $\mathbb{S}$ when several fields may be read during the evaluation of the expression `o2[v]` in a $\mathbb{S}$:

$$|\mathsf{readfields}_{\mathbb{S}}(\texttt{o2}, \texttt{v})| > 1.$$

Example 2.3 (Plural FCT Pattern)**.** We again consider the FCT pattern of the code fragment shown in Figure 2, and assume a concrete pre-condition in which several concrete fields are involved in the pattern. To illustrate such a case, we assume a pre-condition that is quite similar to that of Example 2.2 for a singular FCT pattern: we let object `o2` be the same, but assume that the value of the key variable `v` is either `x` or `y` (it could be only `x` in Example 2.2). Since the two fields `x` and `y` may be read when executing the code, the number of involved fields is two, thus the FCT pattern is plural. Figure 4 presents an analysis of this pattern. The concrete post-condition shows that the fields `x` and `y` of object `o2` are copied to object `o1` with capitalized field names `X` and `Y`, respectively. Again, we present both the concrete and abstract pre- and post-conditions. The abstract post-condition shows that the analysis loses the relations between the origins and destinations of the field copies. Thus, the information in the post-condition about the values of the fields in object `o1` is not as precise as in the concrete level.

Comparison of analyses of FCT patterns We compare analyses of each category in terms of the inference of field correspondence relations.

When a single field is involved in an FCT pattern, the analysis infers that a single field correspondence relation may be observed. On the other hand, when several fields are involved in an

Singular FCT pattern	Plural FCT pattern
Abstract Pre-condition	
o2 $\mapsto$ {x : fx, y : fy} v $\mapsto$ x	o2 $\mapsto$ {x : fx, y : fy} v $\mapsto$ x $\vee$ y

```
t = o2[v];
k = v.toUpperCase();
o1[k] = t;
```

Abstract Post-condition	
o1 $\mapsto$ {X : fx} o2 $\mapsto$ {x : fx, y : fy} v $\mapsto$ x	o1 $\mapsto$ {X : fx $\vee$ fy Y : fx $\vee$ fy} o2 $\mapsto$ {x : fx, y : fy} v $\mapsto$ x $\vee$ y
Field Correspondence Relations	
(o2.x $\bowtie$ o1.X)	(o2.x $\bowtie$ o1.X), (o2.y $\bowtie$ o1.X), (o2.x $\bowtie$ o1.Y), (o2.y $\bowtie$ o1.Y)
Spurious Relations	
none	(o2.y $\bowtie$ o1.X), (o2.x $\bowtie$ o1.Y)

Figure 5. Comparisons of analyses of FCT patterns

FCT pattern, the analysis conservatively approximates that all field correspondence relations made of any combination of the involved fields may occur, which means that spurious field correspondence relations are collected. Since an FCT pattern consists of a series of read and write accesses to object fields, the analysis may lose the relation between the source and destination of a field copy. Thus, the analysis infers a number of spurious field correspondence relations that define infeasible value flows between two objects.

Note that we have assumed so far that variable `o1` —we call it the *destination variable*, as it designates the destination object— stores the address of a single, fully known object. This assumption lets us focus on the spurious field correspondence relations caused in the analysis of plural FCT patterns in the rest of the section. When the destination variable may store the address of several objects, the analysis of a write into one of the fields of this object needs to account for all the possibilities, which induces a *weak update* effect, causing additional imprecision. On the other hand, when the destination object is fully determined, the write operation itself can be handled with a more precise *strong update*. This difference only influences on the analysis of singular FCT patterns, as we have observed that the analysis of plural FCT patterns is imprecise anyway.

We compare the analyses of two Examples 2.2 and 2.3 in terms of the inference of field correspondence relations. Figure 5 shows the comparison of the analyses for each category of FCT patterns. When we consider the pre-condition of the plural FCT pattern, there are two possible executions, namely one for each possible value of variable `v`. For each execution, the fields `x` and `y` are copied to the corresponding fields named `X` and `Y`, respectively. Thus, the field correspondence relations `(o2.y ⋈ o1.X)` and `(o2.x ⋈ o1.Y)` are obviously spurious.

When the values of the fields in object `o2` are function closures and a subsequent program statement performs a call to a function designated by one of the closures, the spurious field correspondence relations in turn incur a spurious callee. The number of spurious callees is proportional to the number of fields in object `o2`. Therefore, such patterns are likely to require the analysis of very large numbers of spurious calls, hereby deteriorating both analysis cost and precision.

The need for a new analysis technique As we have shown with the above comparison, we need a sophisticated analysis technique to infer precise field correspondence relations that do not accumulate spurious relations from the analysis of plural FCT patterns. We discuss how precisely existing techniques can infer field correspondence relations for plural FCT patterns.

When an FCT pattern is enclosed in a loop, its key variable is an index variable of the loop, and an analysis approximates the values of the index variable with a non-singleton range, the FCT

DOI: 10.1002/spe

```
1 var i = arr.length;
2 while ( i-- ) {
3   v = arr[i];
4   o1[v] = o2[v];
5 }
```

(a) Example general loop with an FCT pattern

```
1 for (var v in o2) {
2   o1[v] = o2[v];
3 }
```

(b) Example `for-in` loop with an FCT pattern

Figure 6. Example loops with FCT patterns

pattern would be categorized as a plural FCT pattern. When the analysis does not have any form of sensitivities, it needs to consider the values of the key variable in all the iterations of the loop. If the analysis is able to enumerate all the iterations of the loop and the value of the key variable in each iteration designates a single field, the analysis can reason separately about field correspondence relations for each enumerated iteration. The analysis of each iteration is able to consider a singular FCT pattern, thus the above imprecision does not occur.

This technique is used in the state-of-the-art JavaScript analyzers such as TAJS and SAFE_{LSA}. Figure 6 shows two example loops that contain FCT patterns. For the general loop shown in Figure 6(a), TAJS and SAFE_{LSA} enumerate all the concrete iterations by unrolling the loop body as they need. For the `for-in` loop shown in Figure 6(b), TAJS enumerates all the concrete iterations without any specific order and SAFE_{LSA} enumerates them in the order of field creation, which is used to determine the iteration order of a `for-in` loop in a specific browser. TAJS and SAFE_{LSA} use dynamic trace partitioning to enumerate concrete iterations.

Since many loops in simple JavaScript programs can be enumerated in a series of iterations, this dynamic enumeration approach, also known as a dynamic unrolling approach, has been successful in inferring precise field correspondence relations. On the other hand, these concrete enumeration based techniques cannot precisely analyze loops that cannot be enumerated as a finite number of iterations. We note that the enumeration of iterations of the loops depends on the pre-condition of the loops, thus the computation of a more precise pre-condition may enable the loops to be enumerated as finite numbers of iterations. However, it is not always feasible. In particular, when the set of inputs of a program is infinite, and causes the pre-condition to include states with objects that may have an unbounded number of fields, the computation of a more precise pre-condition is not doable.

In the rest of the paper, we devise a new abstraction that helps the analysis to infer more precise field correspondence relations from FCT patterns. The key idea is to let the analysis of one abstract execution of an FCT pattern `o1[`f`(v)] = `g`(o2[v])` describe a set of concrete executions, such that, in each of these executions, a single common field is involved in the FCT pattern. Since there is only one field involved in each abstract execution, the analysis effectively treats the FCT pattern as a singular FCT pattern.

3. OVERVIEW: OUR APPROACH

We give an overview of our approach with an analysis of a plural FCT pattern. We consider the program in Figure 6(a) assuming that array `arr` is unknown and that object `o2` has two fields of names `x` and `y`. This program carries out a partial copy of the fields of object `o2` into object `o1` depending on the contents of `arr`. Under this pre-condition, the loop cannot be enumerated as a finite number of iterations, thus an analysis that relies only on the dynamic enumeration approach would not be able to reason accurately about field correspondence relations.

Even though `arr` is unknown, the program may copy only the fields present in object `o2`. Thus, only the fields `x` and `y` may be copied into `o1` (for short, we call the field designated by the name `v` "the field `v`" and use this terminology throughout the paper). Therefore, the field correspondence relation at the end of the execution of the loop may be any subset of $\{$`(o2.x` $\bowtie$ `o1.x), (o2.y` $\bowtie$ `o1.y)`$\}$.

DOI: 10.1002/spe

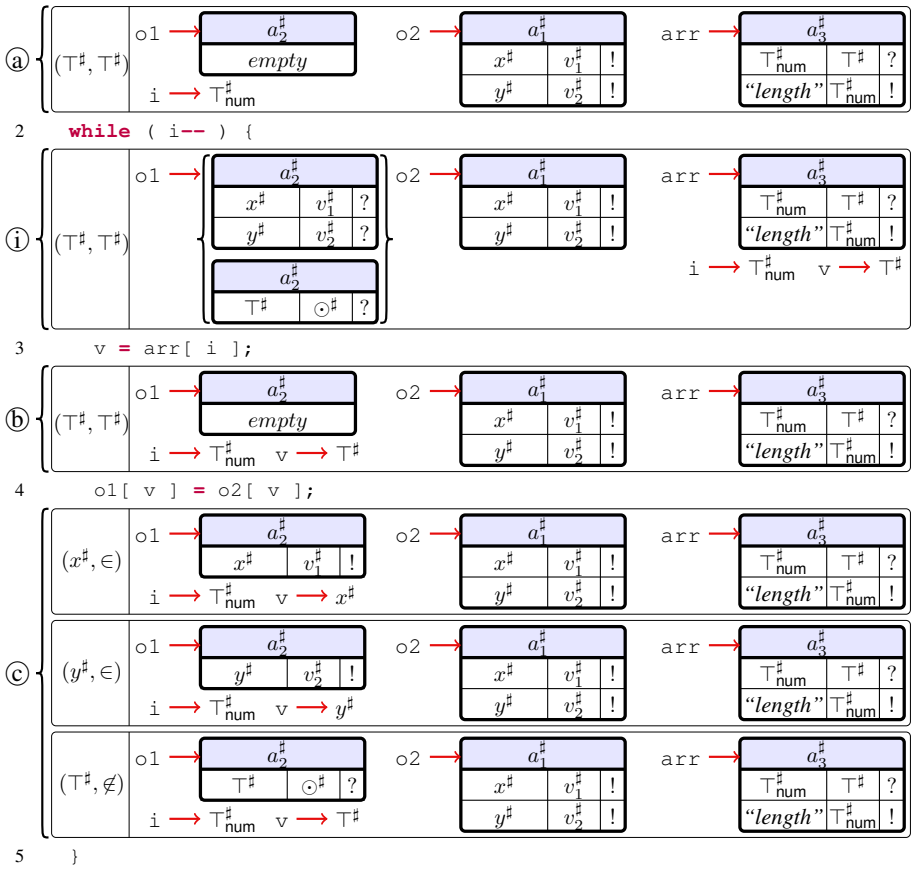


Figure 7. Invariants for the program in Figure 6(a)

3.1. Abstraction

We first study the predicates that the analysis needs to manipulate, and the abstraction that they define. Since the set of addresses that occurs in a program execution is not known statically and is unbounded, the analysis needs to manipulate *summary* abstract addresses that may denote several concrete addresses. The analysis of a write into a summary address needs to account for the effect of a modification to any address it describes, thus it will result in a *weak update*. On the other hand, the analysis also manipulates *non-summary* abstract addresses: such an abstract address denotes a single concrete address [10]. Thus, the analysis of a write into a non-summary address uses a *strong update*, and preserves more precise information than a weak update to a summary address. In this paper, we use the hash symbol to denote *abstract symbols* used in analyses like an abstract address $a^\sharp$. Since the difference between summary and non-summary abstract addresses influences only on the analysis of singular FCT patterns, and since we are interested in plural FCT patterns, we consider only non-summary abstract addresses in the rest of this section.

The analysis often uses a single abstract object to represent several concrete objects that do not have exactly the same set of fields. Therefore, we make an abstract object denote a set of *abstract fields* that may occur in its corresponding concrete objects, and we mark each abstract field with an annotation: "!" (resp., "?") designates its corresponding concrete field *must* (resp., *may*) exist. Since abstract objects describe all the possible fields by abstract fields, a field that is not described by any abstract fields *must not* exist in any corresponding concrete object.

In Figure 7, the abstract states ⓐ, ⓑ, and ⓒ show invariants at each program point in the first iteration over the loop. The state ⓐ shows an initial abstract state, where `o1`, `o2`, and `arr` point to abstract objects at addresses $a_1^\sharp$, $a_2^\sharp$, and $a_3^\sharp$, respectively, which are non-summary abstract addresses.

In the initial state, o1 is empty, and the value of i is an unknown number $\top^\sharp_{\mathsf{num}}$. Object o2 has two abstract fields $x^\sharp$ and $y^\sharp$, and their values are $v_1^\sharp$ and $v_2^\sharp$, respectively. The field names $x^\sharp$ and $y^\sharp$ are non-summary abstract values which respectively denote the strings "x" and "y", and the values $v_1^\sharp$ and $v_2^\sharp$ are the values of the variables v1 and v2. The fields of o2 are annotated with !, as we know that the fields of o2 are exactly $\{\mathtt{x}, \mathtt{y}\}$. Array arr has a definitely existing field "length" and its value is an unknown number. The other abstract field ($\top^\sharp_{\mathsf{num}}$, $\top^\sharp$, ?) expresses that the array has 0 or many elements with unknown values. Note that we write $\odot^\sharp$ to denote the undefined value.

The state ⓘ designates both the loop invariant and the post-condition of the program. We observe that the fields of o1 are annotated with a question mark, which means that they may or may not appear. We note that it captures the intended field correspondence relations (o2.x $\bowtie$ o1.x) and (o2.y $\bowtie$ o1.y): if o1 has a field x, the value of that field is the same as o2.x (and the same for y).

3.2. *Analysis Algorithm*

We now present an analysis to compute invariants such as those shown in Figure 7. It proceeds by forward abstract interpretation [11] using the abstract states described above. It starts with the abstract state ⓑ that describes the initial states: v stores an unknown value $\top^\sharp$ and i stores an unknown number (as arr is unknown). It then progresses through the program forwards, accounts for an over-approximation of the effect of each statement, and computes loop invariants as limits of sequences of abstract iterations over the effect of the loop body.

The first challenge is to analyze precisely the effect of one iteration of the loop body. For the field lookup operation o2[v] at line 4, we expect three possible results: $v_1^\sharp$, $v_2^\sharp$, and undefined because o2 has two abstract fields, $x^\sharp$ and $y^\sharp$, and the lookup operation will return undefined when the field v is not present in the object o2. Since the value of v subsumes the inputs of those three results, and in each case, the loop body has a different effect, the analysis needs to split the states into three sets of states characterized by the result of the lookup operation. In abstract interpretation terms, this means the analysis should perform some *trace partitioning*, so as to reason over the loop body using disjunctive properties. More precisely, the analysis should consider three cases:

1. If v $\in$ o2 and v is $x^\sharp$, since the value of v designates an abstract field of o2 that corresponds to exactly *one* concrete cell denoted by a non-summary address, this abstract field is copied to o1 as a strong update. Therefore, at the end of the loop body, o1 contains an abstract field $(x^\sharp, v_1^\sharp, !)$, as described in the first component $\{(x^\sharp, \in)\}$ of the abstract states ⓒ.
2. The case where v $\in$ o2 and v is $y^\sharp$ is similar (the second component of ⓒ).
3. If v $\notin$ o2, the result of o2[v] is $\odot^\sharp$ since the value of v does not designate a field in o2. Thus, after line 4, o1 may contain abstract fields designated by the value of v whose values are $\odot^\sharp$. The last state $(\top^\sharp, \notin)$ of ⓒ corresponds to this case.

This partitioning operation turns one abstract state into a disjunction of several abstract states that cover the initial abstract state. In our implementation, this partitioning is carried out in two stages: first, the analysis discriminates executions depending on whether v is the name of a field of o2; second, when v is the name of a field of o2, the analysis produces a disjunction over the abstract fields of o2.

The second challenge is to compute precise approximations for unions of sets of states, as *abstract joins* (merging of partitions and control flow paths) and *widening* (union of abstract iterates to let the analysis of loops converge).

We observe that joining partitioned abstract states early would cause a precision loss, so that the analysis needs to delay their join. For example, let us assume that the analysis joins the results that correspond to the three sets of states at the end of the loop body at line 5. Then, the resulting abstract object would become the one in Figure 8, and the third field $(\top^\sharp, \{v_1^\sharp, v_2^\sharp, \odot^\sharp\}, ?)$ means that the value of any field in the object is either $v_1^\sharp$, $v_2^\sharp$, or $\odot^\sharp$; this abstraction fails to capture the precise field correspondence relation. Thus, instead of joining all the analysis results at the end of the loop body, our analysis maintains two abstract objects for the object pointed by o1 as described in ⓘ.

DOI: 10.1002/spe

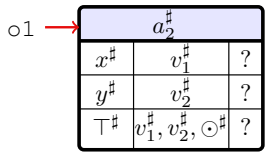


Figure 8. Join of all the results at the end of the loop body in Figure 7

In order to avoid overly large sets of disjuncts caused by delayed joins, the analysis also requires an efficient join algorithm, which is able to merge abstract states while preventing the imprecision shown in Figure 8. To do this, we provide a *join condition* that states when abstract objects can be merged without causing the precision loss discussed above. Essentially, our analysis does not join objects with possibly overlapping abstract fields, when their respective abstract values are not comparable in one direction or the other. For example, at ⓒ, our join algorithm finds a matching of two abstract objects `o1` in $(x^\sharp, \in)$ and $(y^\sharp, \in)$ and they are joined because the abstract field names $x^\sharp$ and $y^\sharp$ are not overlapping. On the contrary, at the same point, our join algorithm does not find any matching for the abstract object `o1` in $(\top^\sharp, \notin)$, and it is not joined with any other because the abstract field name $\top^\sharp$ overlaps with other abstract field names $x^\sharp$ or $y^\sharp$ and its value $\odot^\sharp$ is not comparable with either of the abstract values $v_1^\sharp$ or $v_2^\sharp$.

To ensure the convergence of abstract iterates, a widening operation over abstract objects is necessary. The widening algorithm is based on a similar principle as the join algorithm, but instead of merging two abstract objects, the analysis applies widening to them. Since the widening algorithm also relies on a join condition, it also does not lose the field correspondence relation. For example, at the loop head after the second iteration, the analysis applies widening to the state from the first iteration ⓐ and to the state from the second iteration ⓘ. As the join algorithm does, the widening algorithm also finds two matchings for `o1` in ⓐ and ⓘ: the empty object in ⓐ with each of the objects in ⓘ. Each matching satisfies the join condition because the widening of an empty object with an object does not have the precision loss discussed above. Because each field does not need any widening, the result of widening for those two matchings are the same as the one in `o1` in ⓘ.

4. LANGUAGE: SYNTAX AND SEMANTICS

We define a core language that can define FCT patterns via field lookup and update.

4.1. Syntax

The full syntax and semantics of a core JavaScript language are defined in [2]. To encode the essential behaviors of FCT patterns, we use the following language:

i	::=	${}^l x = e^l$	assignment
	|	${}^l x = \{\}^l$	object initialization
	|	${}^l x = x\,[x]\,^l$	field lookup
	|	${}^l x\,[x] = x^l$	field update
	|	${}^l x = \texttt{iterInit}\,(x)\,^l$	`for-in` loop
	|	${}^l x = \texttt{iterNext}\,(x, x)\,^l$	`for-in` loop
	|	${}^l x = \texttt{iterHasNext}\,(x)\,^l$	`for-in` loop
	|	${}^l \texttt{merge}\,(x)\,^l$	a note for the end of an FCT pattern
	|	${}^l \texttt{while}^l\,(e)\;\; {}^l i^l$	loop
	|	${}^l i^l \texttt{;}\, i^l$	sequence
e	::=	k	string constant
	|	x	variable
	|	$e + e$	string concatenation

DOI: 10.1002/spe

Note that the language has a special instruction `merge` (x) that represents the end of an FCT pattern with the key variable x. We assume that each instruction in a given program has uniquely annotated labels $l \in \mathbb{L}$. We use the labels when we define the concrete semantics of the language as a transition system. For presentation brevity, we omit the labels in other cases. The basic instructions are assignment $x = e$ where the value of expression e is assigned to x, object initialization $x = \{\}$ where a new object is allocated and its address is assigned to x, field lookup $x_1 = x_2\,[\,x_3\,]$ where the value of the field x_3 of object x_2 is stored at x, field update $x_1\,[\,x_2\,] = x_3$ where the value of x_3 is stored at the field x_2 of object x_1, loop `while` (e) i where the loop body i is repeatedly executed while the value of the given condition expression e is a non-empty string, and sequence i_1`;` i_2 where the two instructions i_1 and i_2 are executed sequentially.

The `for-in` loop statement `for` $(y$ `in` $e)$ i is desugared into a sequence of simpler instructions. Since the semantics of the `for-in` statement iterates the fields of the given object e with each field name as the index value y, the instruction $x_1 =$ `iterInit` (x_2) generates a list of fields of object x_2, $x_1 =$ `iterNext` (x_2, x_3) returns the next field to visit, and $x_1 =$ `iterHasNext` (x_2) checks whether any field is not yet visited. For example, the following `for-in` loop:

```
for ( var v in o2 ) { o1[v] = o2[v]; }
```

is roughly desugared into the following code:

```
t = iterInit(o2);
b = iterHasNext(t);
while (b) {
  v = iterNext(o2, t);
  o1[v] = o2[v];
  b = iterHasNext(t);
  merge(v);
}
```

Since the order of the fields to visit in `for-in` loops is not defined in the language semantics [12], we simply assume a non-deterministic order of the fields. The instruction `merge` (x) is a special instruction that represents the end of a loop with the key variable x. The concrete semantics of `merge` is *no operation*.

An expression e is a string constant $k \in \mathbb{V}_{\text{str}}$, a variable $x \in \mathbb{X}$, or string concatenation e_1+e_2.

A concrete *program state* $s \in \mathbb{S}$ is a pair made of a *control state* (unique labels for instructions) $l \in \mathbb{L}$ and a *memory state* $m \in \mathbb{M}$, thus $\mathbb{S} = \mathbb{L} \times \mathbb{M}$. A trace σ is a finite sequence $\langle s_0, ..., s_n \rangle$ where $s_0, ..., s_n \in \mathbb{S}$. We write $\mathbb{S}^\star$ for the set of traces, and let our concrete domain be $(\mathbb{D}, \sqsubseteq) = (\mathcal{P}(\mathbb{S}^\star), \subseteq)$. Also, if σ is a trace, we write $\sigma_\rfloor$ for the last state in σ. A memory state $m \in \mathbb{M} = \mathbb{E} \times \mathbb{H}$ is composed of an environment $\varepsilon \in \mathbb{E} = \mathbb{X} \to \mathbb{V}_{\text{val}}$ which is a map from variables to values and a heap $h \in \mathbb{H} = \mathbb{V}_{\text{addr}} \to \mathbb{O}$ which is a map from addresses to objects. A concrete object $o \in \mathbb{O} = \mathbb{V}_{\text{str}} \to_{\text{fin}} \mathbb{V}_{\text{val}}$ is a map from strings to values. A value $v \in \mathbb{V}_{\text{val}} = \mathbb{V}_{\text{addr}} \uplus \mathbb{V}_{\text{str}}$ is either an address or a string. For the sake of brevity, we abuse the notation for $m(x)$ and $m(a)$ to respectively denote $\varepsilon(x)$ and $h(a)$ where $m = (\varepsilon, h)$ and $a \in \mathbb{V}_{\text{addr}}$.

4.2. Concrete Semantics

We define the semantics of a program i by a transition system $(\mathbb{S}, \tau)$ where $\mathbb{S}$ is a set of program states, and $\tau \subseteq \mathbb{S} \times \mathbb{S}$ is a transition relation. We write $s \to s'$ to denote $(s, s') \in \tau$, meaning that it starts in a state s, and after one step execution, it goes to state s'. The transition relation $\tau[i] \subseteq \mathbb{S} \times \mathbb{S}$ is defined by induction on the syntax of instructions with the semantic function for expressions $E[\![e]\!] : \mathbb{M} \to \mathbb{V}_{\text{val}}$ as follows:

$$
\begin{aligned}
\tau[{}^{l_1}x = e^{l_2}] &\stackrel{\text{def}}{=} \{(l_1, m) \to (l_2, m[x \mapsto v]) \mid m \in \mathbb{M},\ v \in E[\![e]\!](m)\} \\
\tau[{}^{l_1}x = \{\}^{l_2}] &\stackrel{\text{def}}{=} \{(l_1, m) \to (l_2, m[x \mapsto a][a \mapsto \epsilon]) \mid m \in \mathbb{M},\ a \in \mathbb{V}_{\text{addr}},\ a \notin \mathsf{Dom}(m),\ \epsilon \in \mathbb{O}\} \\
\tau[{}^{l_1}x_1 = x_2\,[\,x_3\,]^{l_2}] &\stackrel{\text{def}}{=} \{(l_1, m) \to (l_2, m[x_1 \mapsto v]) \mid m \in \mathbb{M},\ v \in \mathsf{lookup}(m(m(x_2)), m(x_3))\} \\
\tau[{}^{l_1}x_1\,[\,x_2\,] = x_3{}^{l_2}] &\stackrel{\text{def}}{=} \{(l_1, m) \to (l_2, m[m(x_1) \mapsto m(m(x_1))[m(x_2) \mapsto m(x_3)]]) \mid m \in \mathbb{M}\}
\end{aligned}
$$

DOI: 10.1002/spe

$$\tau[^{l_1} x_1 = \texttt{iterInit}\,(x_2)\,^{l_2}] \stackrel{\text{def}}{=}$$
$$\{(l_1, m) \to (l_2, m[x_1 \mapsto a][a \mapsto m(m(x_2))]) \mid m \in \mathbb{M},\ a \in \mathbb{V}_{\text{addr}},\ a \notin \mathsf{Dom}(m)\}$$
$$\tau[^{l_1} x_1 = \texttt{iterNext}\,(x_2, x_3)\,^{l_2}] \stackrel{\text{def}}{=}$$
$$\{(l_1, m) \to (l_2, m[m(x_3) \mapsto o \setminus k][x_1 \mapsto k]) \mid m \in \mathbb{M},\ o \in m(m(x_3)),\ k \in \mathsf{fields}(o)\}$$
$$\tau[^{l_1} x_1 = \texttt{iterHasNext}\,(x_2)\,^{l_2}] \stackrel{\text{def}}{=}$$
$$\{(l_1, m) \to (l_2, m[x_1 \mapsto \text{“}true\text{”}]) \mid m \in \mathbb{M},\ \exists k \in \mathbb{V}_{\text{str}},\ k \in \mathsf{Dom}(m(m(x_2)))\} \cup$$
$$\{(l_1, m) \to (l_2, m[x_1 \mapsto \text{“”}]) \mid m \in \mathbb{M},\ \forall k \in \mathbb{V}_{\text{str}},\ k \notin \mathsf{Dom}(m(m(x_2)))\}$$
$$\tau[^{l_1}\texttt{merge}\,(x)\,^{l_2}] \stackrel{\text{def}}{=} \{(l_1, m) \to (l_2, m)\}$$
$$\tau[^{l_1}\texttt{while}^{l_2}\,(e)\ \ ^{l_3} i^{l_4}] \stackrel{\text{def}}{=} \{(l_1, m) \to (l_2, m) \mid m \in \mathbb{M}\} \cup$$
$$\{(l_2, m) \to (l_3, m) \mid m \in \mathbb{M},\ \text{“”} \notin E[\![e]\!](m)\} \cup$$
$$\{(l_2, m) \to (l_4, m) \mid m \in \mathbb{M},\ \text{“”} \in E[\![e]\!](m)\} \cup \tau[^{l_3} i^{l_2}]$$
$$\tau[^{l_1} i_1^{l_2}\texttt{;}\, i_2\ ^{l_3}] \stackrel{\text{def}}{=} \tau[^{l_1} i_1^{l_2}] \cup \tau[^{l_2} i_2^{l_3}]$$

$$E[\![k]\!](m) \stackrel{\text{def}}{=} k \quad \text{where } k \in \mathbb{V}_{\text{str}}$$
$$E[\![x]\!](m) \stackrel{\text{def}}{=} m(x)$$
$$E[\![e_1\texttt{+}e_2]\!](m) \stackrel{\text{def}}{=} k_1 \cdot k_2 \quad \text{where } k_1 \in E[\![e_1]\!](m) \text{ and } k_2 \in E[\![e_2]\!](m)$$

where the auxiliary function $\mathsf{lookup} : \mathbb{O} \times \mathbb{V}_{\text{str}} \to \mathbb{V}_{\text{val}}$ returns a field value designated by a given string in a given object or the `undefined` value when there is no such a field in the object, the auxiliary function $\mathsf{fields} : \mathbb{O} \to \mathbb{V}_{\text{str}}$ returns an arbitrary field name associated to a given object, and $\epsilon \in \mathbb{O}$ denotes an empty object. The assignment instruction $x = e$ assigns the value of e to x. The object initialization instruction $x = \{\}$ creates a new object and assigns its address to x. The field lookup instruction $x_1 = x_2\,[\,x_3\,]$ assigns the field value of object x_2 designated by the value of x_3 to x_1. The field update instruction $x_1\,[\,x_2\,] = x_3$ updates the field of object x_1 designated by the value of x_2 with the value of x_3. The instruction $x_1 = \texttt{iterInit}\,(x_2)$ generates a list of field names of the given object x_2 and assigns it to x_1 as an object. The instruction $x_1 = \texttt{iterNext}\,(x_2)$ looks up a next field name to visit from object x_2 and assigns the next field name to x_1. The instruction $x_1 = \texttt{iterHasNext}\,(x_2)$ checks whether there is a next field name to visit in object x_2. If it exists, the instruction assigns a non-empty string to x_1; otherwise, it assigns an empty string to x_1. We write $\mathsf{Dom}(m)$ and $\mathsf{Dom}(o)$ for the set of addresses of a memory m and the set of fields of an object o, respectively. We write $o \setminus s$ to denote a field deletion operation that removes the field named s from object o. The $\texttt{while}\,(e)\ i$ instruction repeats the loop body i until the given loop condition e is not satisfied. The value expression k is evaluated as a constant string. The variable expression x is evaluated as the value of variable x. The expression $e_1\texttt{+}e_2$ is evaluated as a concatenation of two values of the expressions e_1 and e_2. We extend the concrete semantic function to handle a set of states: $[\![i]\!]_C(S) = \{\sigma \cdot s \mid \exists \sigma \in S,\ \sigma_{\rfloor} \to s\}$.

4.3. Field Correspondence Relations

Now, we formalize the definition of *field correspondence relations*. We utilize a non-standard semantics that annotates each object field with where it has been copied from. This allows us to use a semantics expressed in terms of reachable states at the cost of keeping partial program executions in the states. To this end, we augment fields with their origin information.

A concrete origin $g \in \mathbb{G}$ stores the location a field was copied from. A concrete environment $\overline{\varepsilon} \in \overline{\mathbb{E}}$ maps each variable to a pair of its value and origins. Other domains are augmented likewise:

$\mathbb{G} = \mathbb{V}_{\text{addr}} \times \mathbb{V}_{\text{str}}$	concrete origins	
$\overline{\mathbb{S}} = \mathbb{L} \times \overline{\mathbb{M}}$	program states	} augmented with $\mathbb{G}$
$\overline{\mathbb{M}} = \overline{\mathbb{E}} \times \overline{\mathbb{H}}$	memory states	
$\overline{\mathbb{E}} = \mathbb{X} \to_{\mathsf{fin}} \mathbb{V}_{\text{val}} \times \mathcal{P}(\mathbb{G})$	environments	
$\overline{\mathbb{H}} = \mathbb{V}_{\text{addr}} \to \overline{\mathbb{O}}$	heaps	
$\overline{\mathbb{O}} = \mathbb{V}_{\text{str}} \to_{\mathsf{fin}} \mathbb{V}_{\text{val}} \times \mathcal{P}(\mathbb{G})$	objects	

These augmented states induce an augmented transition relation $\overline{\tau}[i] \subseteq \overline{\mathbb{S}} \times \overline{\mathbb{S}}$, which is defined with the augmented semantic function for expressions $\overline{E}[\![e]\!] : \overline{\mathbb{M}} \to \mathbb{V}_{\text{val}} \times \mathcal{P}(\mathbb{G})$:

$$\overline{\tau}[^{l_1} x_1 = x_2\,[\,x_3\,]^{\,l_2}] \stackrel{\text{def}}{=} \{(l_1, m) \to (l_2, m[x_1 \mapsto (v, \{(a,k)\})]) \,|\, m \in \mathbb{M},\ (a,\_) \in m(x_2),\ o \in m(a),\ k \in m(x_3),\ v \in \mathsf{lookup}(o,k)\}$$

$$\overline{E}[\![k]\!](m) \stackrel{\text{def}}{=} (k, \emptyset) \ \text{ where } \ k \in \mathbb{V}_{\text{str}}$$
$$\overline{E}[\![e_1{+}e_2]\!](m) \stackrel{\text{def}}{=} (k_1 \cdot k_2, G_1 \cup G_2) \ \text{ where } \ (k_1, G_1) \in \overline{E}[\![e_1]\!](m), (k_2, G_2) \in \overline{E}[\![e_2]\!](m)$$

The other cases are the same.

We also extend the augmented semantic function to handle a set of augmented states as follows:

$$\begin{aligned} \overline{[\![i]\!]}_C : \mathcal{P}(\overline{\mathbb{S}}^\star) &\longrightarrow \mathcal{P}(\overline{\mathbb{S}}^\star) \\ \overline{S} &\longmapsto \{\overline{\sigma} \cdot \overline{s} \,|\, \exists \overline{\sigma} \in \overline{S},\ \overline{\sigma}_{\rfloor} \to_{\mathsf{aug}} \overline{s}\}. \end{aligned}$$

where $\overline{s} \to_{\mathsf{aug}} \overline{s}'$ denotes $(\overline{s}, \overline{s}') \in \overline{\tau}$.

We can define an abstract relation $\alpha_C : \mathcal{P}(\overline{\mathbb{S}}^\star) \to \mathcal{P}(\mathbb{S}^\star)$ from the augmented semantics to the standard semantics as follows:

$$\begin{aligned} \alpha_C : \mathcal{P}(\overline{\mathbb{S}}^\star) &\longmapsto \mathcal{P}(\mathbb{S}^\star) \\ \overline{S} &\longmapsto \{\langle \psi(\overline{s}_0), ..., \psi(\overline{s}_n)\rangle \,|\, \langle \overline{s}_0, ..., \overline{s}_n\rangle \in \overline{S}\} \end{aligned}$$

where $\psi : \overline{\mathbb{S}} \to \mathbb{S}$ is a function that removes the augmented information from a given state.

Then, properties described by the augmented semantics imply properties described by the standard semantics.

Theorem 4.1 (Implication of Abstraction Relation)**.** For a given program i, the program semantics characterized by the least fixpoint of the augmented semantic function has a corresponding program trace in the program semantics characterized by the least fixpoint of the standard semantic function:

$$\text{lfp}[\![i]\!]_C = \alpha_C(\text{lfp}\overline{[\![i]\!]}_C).$$

We can now formally define field correspondence relations:

Definition 4.1 (Field Correspondence Relations)**.** A state $\overline{s} \in \overline{\mathbb{S}}$ augmented with field correspondence relations admits a field correspondence relation $(\overline{o}_0.k_0 \bowtie \overline{o}_1, k_1)$ for objects $\overline{o}_0, \overline{o}_1 \in \phi(\overline{s})$ and strings $k_0, k_1 \in \mathbb{V}_{\text{str}}$ if and only if the following condition holds:

$$\exists v \in \mathbb{V}_{\text{val}},\ \exists X \subseteq \mathbb{G},\ \overline{o}_1(k_1) = (v, X) \ \wedge\ (a, k_0) \in X \ \wedge\ \overline{s}(a) = \overline{o}_0$$

where $\phi(\overline{s}) \longmapsto \{\overline{o} \,|\, a \in \mathbb{V}_{\text{addr}}, \overline{o} \in \overline{s}(a)\}$ collects a set of objects in a given state $\overline{s}$.

Example 4.1 (Field Correspondence Relations)**.** We consider the following simple code fragment that makes field correspondence relations from object `o1` to object `o2`:

```
1   v1 = o1[x];
2   v2 = o1[y];
3   o2[z] = v1 + v2;
```

where variables `x`, `y`, and `z` respectively denote string constants x, y, and z. At line 1, by the augmented semantic function for a field lookup instruction, variable `v1` keeps a value from the field of object `o1` and its origin information $\{a, x\}$ where a is an address pointing to object `o1` and x is the field name where the value loaded from. Similarly, variable `v2` keeps a value and its origin information $\{a, y\}$ at line 2. Since the origin information from `v1` and `v2` are merged at line 3 by the augmented semantic function for string concatenation, the state at the end of the program admits the field correspondence relations `(o1.x` $\bowtie$ `o2.z)` and `(o1.y` $\bowtie$ `o2.z)`.

We introduce the field correspondence relations as a tool to measure analysis precision of FCT patterns. Since we have shown how our abstraction can precisely infer the field correspondence relations in Section 3, we define abstractions based on the standard semantics in the rest of the paper.

DOI: 10.1002/spe

5. COMPOSITE ABSTRACTION

In this section, we formally define a composite abstraction to reason over FCT patterns and to infer precise field correspondence relations. The composite abstraction consists of three components: a partitioning abstraction, an object abstraction, and a string abstraction. The partitioning abstraction supports separate reasoning about the executions in an FCT pattern. The object abstraction describes accurate properties over fields, so as to disambiguate the field correspondence relations even after abstract joins. Lastly, the string abstraction describes object field names, so as to disambiguate them.

Structure of the abstraction. Our composite abstraction is defined on top of a standard abstraction for JavaScript heaps. We now define this abstraction and fix the notations that we use throughout the rest of the paper. Essentially, the abstraction of traces is built on top of an abstraction of memory states in Section 5.1. An abstract memory is made of an abstract environment that maps variables to abstract values and of an abstract heap that maps abstract addresses to abstract objects. An abstract value comprises of an abstraction for a set of strings and an abstraction for a set of addresses. Abstract objects are defined in Section 5.2 and abstract strings in Section 5.3. To summarize, the abstract domains are defined as follows:

$\mathbb{M}^\sharp$	$=$	$\mathbb{E}^\sharp \times \mathbb{H}^\sharp$	abstract memory states
$\mathbb{E}^\sharp$	$=$	$\mathbb{X} \to_{\text{fin}} \mathbb{V}^\sharp$	abstract environments
$\mathbb{H}^\sharp$	$=$	$\mathbb{V}^\sharp_{\text{addr}} \to \mathbb{O}^\sharp$	abstract heaps
$\mathbb{O}^\sharp$	:		abstract objects defined in Section 5.2 based on $\mathbb{V}^\sharp_{\text{str}}, \mathbb{V}^\sharp$
$\mathbb{V}^\sharp$	$=$	$\mathbb{V}^\sharp_{\text{str}} \times \mathbb{V}^\sharp_{\text{addr}}$	abstract values
$\mathbb{V}^\sharp_{\text{str}}$	:		abstract strings defined in Section 5.3
$\mathbb{V}^\sharp_{\text{addr}}$	:		abstract addresses, parameter of the analysis
$\mathbb{D}^\sharp$	:		abstract traces defined in Section 5.1 based on $\mathbb{M}^\sharp$

We assume that the concretization function $\gamma_{\text{addr}} : \mathbb{V}^\sharp_{\text{addr}} \to \mathcal{P}(\mathbb{V}_{\text{addr}})$ for the address abstract domain is given. Furthermore, the concretization function for strings $\gamma_{\text{str}} : \mathbb{V}^\sharp_{\text{str}} \to \mathcal{P}(\mathbb{V}_{\text{str}})$, the concretization function for object $\gamma_{\mathbb{O}^\sharp} : \mathbb{O}^\sharp \to \mathcal{P}(\mathbb{O})$, and the concretization function for traces $\gamma : \mathbb{D}^\sharp \to \mathbb{D}$ are defined together with the corresponding abstract domains in the rest of this section. Based on these, the abstractions of the domains that are already fully specified are as follows:

$$
\begin{array}{lll}
\gamma_{\mathbb{V}^\sharp} : \mathbb{V}^\sharp & \longrightarrow & \mathcal{P}(\mathbb{V}_{\text{val}}) \\
\quad (k^\sharp, a^\sharp) & \longmapsto & \gamma_{\text{str}}(k^\sharp) \cup \gamma_{\text{addr}}(a^\sharp) \\
\gamma_{\mathbb{E}^\sharp} : \mathbb{E}^\sharp & \longrightarrow & \mathcal{P}(\mathbb{E}) \\
\quad \varepsilon^\sharp & \longmapsto & \{\varepsilon \in \mathbb{E} \mid \forall x \in \mathbb{X},\ \varepsilon(x) \in \gamma_{\mathbb{V}^\sharp}(\varepsilon^\sharp(x))\} \\
\gamma_{\mathbb{H}^\sharp} : \mathbb{H}^\sharp & \longrightarrow & \mathcal{P}(\mathbb{H}) \\
\quad h^\sharp & \longmapsto & \{h \in \mathbb{H} \mid \forall a^\sharp \in \mathbb{V}^\sharp_{\text{addr}},\ a \in \gamma_{\text{addr}}(a^\sharp),\ h(a) \in \gamma_{\mathbb{O}^\sharp}(h^\sharp(a^\sharp))\} \\
\gamma_{\mathbb{M}^\sharp} : \mathbb{M}^\sharp & \longrightarrow & \mathcal{P}(\mathbb{M}) \\
\quad (e^\sharp, h^\sharp) & \longmapsto & \{(e, h) \mid e \in \gamma_{\mathbb{E}^\sharp}(e^\sharp), h \in \gamma_{\mathbb{H}^\sharp}(h^\sharp)\}.
\end{array}
$$

5.1. Partitioning Abstraction

A trace partitioning abstraction (or, for short, partitioning abstraction) splits a set of traces based on a specific trace abstraction so as to reason more accurately on smaller sets of executions. We first set up a general partitioning framework, then define two instances with two trace abstractions that provide two stages of partitioning we used in Section 3. We use the notations for traces introduced in Section 4.1.

In the following, a trace partitioning abstraction is an instance of the following definition:

Definition 5.1 (Partitioning Abstraction)**.** We let $\mathbb{T}^\sharp$ and $\gamma_{\mathbb{T}^\sharp} : \mathbb{T}^\sharp \to \mathbb{D}$ define a trace abstraction that is covering (i.e., $\mathbb{D} = \cup\{\gamma_{\mathbb{T}^\sharp}(t^\sharp) \mid t^\sharp \in \mathbb{T}^\sharp\}$) and we use the store abstraction $\mathbb{M}^\sharp$ and $\gamma_{\mathbb{M}^\sharp} : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M})$ defined above. Then, the *partitioning abstraction* parameterized by these two abstractions is

Softw. Pract. Exper. (0000)
DOI: 10.1002/spe

defined by the domain $\mathbb{D}^\sharp = \mathbb{T}^\sharp \to \mathbb{M}^\sharp$ and the concretization function γ defined by:

$$\begin{array}{rccl} \gamma: & \mathbb{D}^\sharp & \longrightarrow & \mathbb{D} \\ & X^\sharp & \longmapsto & \{\sigma \in \mathbb{S}^\star \mid \forall t^\sharp \in \mathbb{T}^\sharp,\ \sigma \in \gamma_{\mathbb{T}^\sharp}(t^\sharp) \Rightarrow \sigma_{\rfloor M} \in \gamma_{\mathbb{M}^\sharp}(X^\sharp(t^\sharp))\} \end{array}$$

where $\sigma_{\rfloor M}$ denotes the memory state of the last state in σ.

Intuitively, this abstraction lets us tie abstract store properties (described by elements of $\mathbb{M}^\sharp$) to abstract traces properties (described by elements of $\mathbb{T}^\sharp$).

Trace abstraction by field existence. The first instance of the partitioning abstraction abstracts traces depending on the existence or absence of a field in an object. It is based on the following trace abstraction:

Definition 5.2 (Abstract Trace Partitioning based on Field Existence)**.** The trace abstraction $\mathbb{T}^\sharp_{\mathsf{fe}}$ is defined by the following abstract elements and abstraction relations:

- a quadruple $(l, \mathtt{x}, \mathtt{y}, p)$, where $l \in \mathbb{L}$, $\mathtt{x}$ is a variable, $\mathtt{y}$ is a variable (pointing to an object) and $p \in \{\in, \notin\}$, denotes all the traces that go through the control state l with a memory state such that the membership of the field, the name of which is the value of $\mathtt{x}$, in the object pointed to by $\mathtt{y}$ is characterized by p;
- the element $\top^\sharp$ abstracts any trace.

The concretization of an abstract trace is defined as follows:

$$\begin{array}{rccl} \gamma_{\mathbb{T}^\sharp_{\mathsf{fe}}}: \mathbb{T}^\sharp_{\mathsf{fe}} & \longrightarrow & \mathcal{P}(\mathbb{S}^\star) \\ (l, \mathtt{x}, \mathtt{y}, \in) & \longmapsto & \mathbb{S}^\star \cdot \{\langle (l, m)\rangle \mid m \in \mathbb{M}, m(\mathtt{x}) \in \phi(m(m(\mathtt{y})))\} \cdot \{\langle (l', m)\rangle \mid l \neq l', m \in \mathbb{M}\}^\star \\ (l, \mathtt{x}, \mathtt{y}, \notin) & \longmapsto & \mathbb{S}^\star \cdot \{\langle (l, m)\rangle \mid m \in \mathbb{M}, m(\mathtt{x}) \notin \phi(m(m(\mathtt{y})))\} \cdot \{\langle (l', m)\rangle \mid l \neq l', m \in \mathbb{M}\}^\star \\ \top^\sharp & \longmapsto & \mathbb{S}^\star \end{array}$$

where $\phi(o) \longmapsto \{k \mid k \in \mathbb{V}_{\mathrm{str}}, v \in o(k)\}$ collects a set of field names in a given object o, and $o(\cdot) : \mathbb{V}_{\mathrm{str}} \to \mathbb{V}_{\mathrm{val}}$ returns the value of a given field name of object o.

Example 5.1. As an example, we consider the following JavaScript code fragment:

```
1  t = o1[x]_l;
```

where the variable `o1` designates an object that has a single field `a`, the value of variable `x` is unknown, and l is a control state after the first statement. The abstract traces $(l, \mathtt{x}, \mathtt{o1}, \in)$ and $(l, \mathtt{x}, \mathtt{o1}, \notin)$ describe all the traces that can be observed at the end of the code fragment. The abstract trace $(l, \mathtt{x}, \mathtt{o1}, \in)$ denotes the set of traces such that the value of variable `x` designates a field in `o1` at the control state l, whereas the abstract trace $(l, \mathtt{x}, \mathtt{o1}, \notin)$ describes the set of traces such that the object `o1` does not have a field designated by the value of variable `x` at the control state l.

Trace abstraction by variable value. The second instance of the partitioning abstraction discriminates traces based on the value of a given variable at a specific control point. To describe information about a specific variable, this partitioning abstraction is parameterized by a value abstraction defined by a domain $\mathbb{V}^\sharp$ and a concretization function $\gamma_{\mathbb{V}^\sharp} : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V}_{\mathrm{val}})$:

Definition 5.3 (Abstract Trace Partitioning based on Variable Value)**.** The trace abstraction $\mathbb{T}^\sharp_{\mathsf{vv}}$ is defined by the following abstract elements and abstraction relations:

- a triple $(l, \mathtt{x}, v^\sharp)$, where $l \in \mathbb{L}$, $\mathtt{x}$ is a program variable, and $v^\sharp$ is an abstract value $v^\sharp \in \mathbb{V}^\sharp$, describes the traces that visit l with a memory state such that the value of $\mathtt{x}$ can be described by $v^\sharp$;
- the element $\top^\sharp$ abstracts any trace.

Then, the concretization of an abstract trace is defined as follows:

$$\begin{array}{rccl} \gamma_{\mathbb{T}^\sharp_{\mathsf{vv}}}: \mathbb{T}^\sharp_{\mathsf{vv}} & \longrightarrow & \mathcal{P}(\mathbb{S}^\star) \\ (l, \mathtt{x}, v^\sharp) & \longmapsto & \mathbb{S}^\star \cdot \{\langle (l, m)\rangle \mid m \in \mathbb{M}, m(\mathtt{x}) \in \gamma_{\mathbb{V}^\sharp}(v^\sharp)\} \cdot \{\langle (l', m)\rangle \mid l \neq l', m \in \mathbb{M}\}^\star \\ \top^\sharp & \longmapsto & \mathbb{S}^\star \end{array}$$

DOI: 10.1002/spe

Example 5.2. As an example, we consider the same JavaScript code fragment as in Example 5.1:

```
1  t = o1[x]_l;
```

We recall that variable `o1` designates an object that has a single field `a`, that the value of variable `x` is assumed to be unknown, and that l stands for a control state after the first statement. The abstract element $(l, \mathtt{x}, \top^\sharp)$ denotes the set of traces such that the value of variable `x` is unknown at the given control state l. This corresponds to all the traces that reach the end of this code fragment.

Combination of trace abstractions. Trace abstractions can be combined into a reduced product abstraction, so as to tie abstract store properties to conjunctions of abstract trace properties. This leads to the formal definition of the two-stage partitioning abstraction discussed in Section 3:

Definition 5.4 (Combination of Abstract Traces)**.** An abstract trace characterized by a combination of two abstract traces is a reduced product [13] of their trace abstractions.

Thus, the combination of two abstract traces is defined by:

$$\mathbb{T}^\sharp \stackrel{\text{def}}{=} \mathbb{T}^\sharp_{\mathsf{fe}} \times \mathbb{T}^\sharp_{\mathsf{vv}}.$$

Moreover, its concretization is defined as follows:

$$\gamma_{\mathbb{T}^\sharp}((l, \mathtt{x}, \mathtt{y}, p), (l, \mathtt{x}, v^\sharp)) \stackrel{\text{def}}{=} \{\sigma \mid \sigma \in \gamma_{\mathbb{T}^\sharp_{\mathsf{fe}}}(l, \mathtt{x}, \mathtt{y}, p),\ \sigma \in \gamma_{\mathbb{T}^\sharp_{\mathsf{vv}}}(l, \mathtt{x}, v^\sharp)\}$$

where $\sigma \in \mathbb{S}^\star$ is a trace.

Our analysis uses $\mathbb{T}^\sharp \stackrel{\text{def}}{=} \mathbb{T}^\sharp_{\mathsf{fe}} \times \mathbb{T}^\sharp_{\mathsf{vv}}$, which means that it ties store abstractions to characterizations of traces based both on field existence and variable values.

Example 5.3. We consider the same JavaScript code fragment as in Example 5.1 and in Example 5.2:

```
1  t = o1[x]_l;
```

The product partitioning abstraction divides the execution traces that can be observed at the end of this code fragment into two sets of traces depending on the result of the statement at line 1. Indeed, there are two possible results on the statement: either variable `x` designates the field `a` in object `o1` or it does not. Abstract traces $((l, \mathtt{x}, \mathtt{o1}, \in), (l, \mathtt{x}, \mathtt{a}))$ and $((l, \mathtt{x}, \mathtt{o1}, \notin), (l, \mathtt{x}, \top^\sharp))$ respectively describe these two sets of traces. The abstract trace $((l, \mathtt{x}, \mathtt{o1}, \in), (l, \mathtt{x}, \mathtt{a}))$ denotes the set of traces such that the lookup expression in the first statement returns the value of field `a` in object `o1` because two components of the abstract trace describe that

- $(l, \mathtt{x}, \mathtt{o1}, \in)$: object `o1` has a field designated by the value of variable `x`, and
- $(l, \mathtt{x}, \mathtt{a})$: the value of variable `x` is `a`.

The other abstract trace $((l, \mathtt{x}, \mathtt{o1}, \notin), (l, \mathtt{x}, \top^\sharp))$ denotes the set of traces such that the lookup expression fails to lookup a field in object `o1` and returns the `undefined` value because two components describe that

- $(l, \mathtt{x}, \mathtt{o1}, \notin)$: object `o1` does not have a field designated by the value of variable `x`, and
- $(l, \mathtt{x}, \top^\sharp)$: the value of variable `x` is unknown.

Example 5.4. We recall the example that we have shown in Section 3:

```
1  var i = arr.length_l0;
2  while ( i-- ) {
3    v = arr[i]_l1;
4    o1[v] = o2[v]_l2;
5  }_l3
```


DOI: 10.1002/spe

$$
\begin{array}{rrcl}
\gamma_{\mathbb{O}^\sharp}: & \mathbb{O}^\sharp & \longrightarrow & \mathcal{P}(\mathbb{O}) \\
& l & \longmapsto & \{o \in \mathbb{O} \mid o \in \gamma_l(l),\ \mathsf{Dom}(o) \subseteq \mathsf{Dom}^\sharp_l(l)\} \\
\gamma_l: & p & \longmapsto & \gamma_p(p) \\
& l_0 \wedge l_1 & \longmapsto & \gamma_l(l_0) \cap \gamma_l(l_1) \\
& l_0 \vee l_1 & \longmapsto & \gamma_l(l_0) \cup \gamma_l(l_1) \\
\gamma_p: & \epsilon & \longmapsto & \mathbb{O} \\
& (k^\sharp \mapsto !, v^\sharp) & \longmapsto & \{o \in \mathbb{O} \mid \forall k \in \gamma_{\mathrm{str}}(k^\sharp),\ o(k) \in \gamma_{\mathbb{V}^\sharp}(v^\sharp)\} \\
& (k^\sharp \mapsto ?, v^\sharp) & \longmapsto & \{o \in \mathbb{O} \mid \forall k \in \gamma_{\mathrm{str}}(k^\sharp),\ k \in \mathsf{Dom}(o) \Rightarrow o(k) \in \gamma_{\mathbb{V}^\sharp}(v^\sharp)\}
\end{array}
$$

$$
\begin{array}{rrcl}
\mathsf{Dom}^\sharp_l: & p & \longmapsto & \mathsf{Dom}^\sharp_p(p) \\
& l_0 \wedge l_1 & \longmapsto & \mathsf{Dom}^\sharp_l(l_0) \cup \mathsf{Dom}^\sharp_l(l_1) \\
& l_0 \vee l_1 & \longmapsto & \mathsf{Dom}^\sharp_l(l_0) \cup \mathsf{Dom}^\sharp_l(l_1)
\end{array}
\qquad
\begin{array}{rrcl}
\mathsf{Dom}^\sharp_p: & \epsilon & \longmapsto & \emptyset \\
& (k^\sharp \mapsto E, v^\sharp) & \longmapsto & \gamma_{\mathrm{str}}(k^\sharp)
\end{array}
$$

Figure 9. Concretization of abstract objects

We assume that the values initially stored in the array `arr` are unknown, and let l_i be the control states after the i-th statement. By applying our trace partitioning abstraction only to the lookup expression at line 4, we observe three trace abstract properties after this statement, as shown at point ⓒ in Figure 7. The abstract properties correspond to the partitions which are defined by $((l_2, \mathtt{v}, \mathtt{o1}, \in), (l_2, \mathtt{v}, x^\sharp))$, $((l_2, \mathtt{v}, \mathtt{o1}, \in), (l_2, \mathtt{v}, y^\sharp))$, and $((l_2, \mathtt{v}, \mathtt{o1}, \notin), (l_2, \mathtt{v}, \top^\sharp))$.

- $((l_2, \mathtt{v}, \mathtt{o1}, \in), (l_2, \mathtt{v}, x^\sharp))$ denotes the set of traces such that an object `o1` has a field designated by the value of variable `v` and the value of variable `v` is described by abstract value $x^\sharp$ at a given control state l_2.
- $((l_2, \mathtt{v}, \mathtt{o1}, \notin), (l_2, \mathtt{v}, \top^\sharp))$ denotes the set of traces such that an object `o1` does not have a field designated by the value of variable `v` and the value of variable `v` was unknown at a given control state l_2.

5.2. Object Abstraction

The object abstraction forms the second layer of our composite abstraction, and describes the fields of JavaScript objects and their contents. A JavaScript object consists of a (non-fixed) set of mutable fields. The main operations on an object are field lookup, field update, and field deletion. A concrete object is a map $o \in \mathbb{O} = \mathbb{V}_{\mathrm{str}} \rightarrow_{\mathsf{fin}} \mathbb{V}_{\mathrm{val}}$.

To accurately handle absent fields, our object abstraction tracks cases where a field is definitely absent. Similarly, to handle lookup of definitely existing fields precisely, the object abstraction annotates such fields with the predicate "!". Fields that may or may not exist are annotated with the predicate "?". The lookup of such a field may return a value or the `undefined` value. Following the conventions set in the beginning of the section, we use an abstraction for values that is defined by the domain $\mathbb{V}^\sharp$ and the concretization function $\gamma_{\mathbb{V}^\sharp} : \mathbb{V}^\sharp \rightarrow \mathcal{P}(\mathbb{V}_{\mathrm{val}})$. Moreover, we rely on an abstraction for string values that is defined by the finite height domain $\mathbb{V}^\sharp_{\mathrm{str}}$ and the concretization function $\gamma_{\mathrm{str}} : \mathbb{V}^\sharp_{\mathrm{str}} \rightarrow \mathcal{P}(\mathbb{V}_{\mathrm{str}})$.

Definition 5.5 (Abstract Object)**.** An *abstract object* $o^\sharp \in \mathbb{O}^\sharp$ is a logical formula described by the following grammar:

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
o^\sharp & ::= & l & l & ::= & p \mid l \wedge l \mid l \vee l & k^\sharp & \in & \mathbb{V}^\sharp_{\mathrm{str}} \\
p & ::= & \epsilon \mid (k^\sharp \mapsto E, v^\sharp) & E & ::= & ! \mid ? & v^\sharp & \in & \mathbb{V}^\sharp
\end{array}
$$

The concretization of $\gamma_{\mathbb{O}^\sharp}(o^\sharp)$ of an abstract object $o^\sharp$ is defined in Figure 9.

An abstract field p is either ϵ, which represents an object with unknown fields, or $(k^\sharp \mapsto E, v^\sharp)$, which represents any object such that the membership of fields the names of which are represented

by $k^\sharp$ is described by E and the values of the corresponding fields (if any) by $v^\sharp$. Note that E may be ! (must exist) or ? (may exist). A logical formula l describes a set of objects, either as an abstract field p, or as a union or intersection of sets of objects. A conjunction of abstract fields represents a set of concrete JavaScript objects. A disjunction $l_0 \vee l_1$ represents two abstract objects l_0 and l_1 as for object `o2` in the abstract state at point ⓘ in Figure 7. Because the objects represented by an abstract field p may have an unbounded number of fields, the concretization of abstract objects $\gamma_{\mathbb{O}^\sharp}$ bounds the set of fields with $\mathsf{Dom}(o) \subseteq \mathsf{Dom}^\sharp_l(l)$. The auxiliary function $\mathsf{Dom}^\sharp_l(l)$ represents a set of fields that are defined in a given logical formula l, which expresses that a field is definitely absent if the field is not defined in l. For example, the concretization of an abstract field $\gamma_p((x^\sharp \mapsto !, v^\sharp))$ represents a set of objects, where the values of the fields designated by the abstract string $x^\sharp$ are represented by the abstract value $v^\sharp$ and the other fields are unknown, whereas the abstract object concretization $\gamma_{\mathbb{O}^\sharp}((x^\sharp \mapsto !, v^\sharp))$ represents the set of objects such that the names of all fields are described by $x^\sharp$, that is any field, the name of which is not described by $x^\sharp$, is absent from the object. The conjunction and disjunction have the usual meaning.

Example 5.5. We consider the following JavaScript code fragment:

```
1 if ( c ) {
2   o[ab] = "x";
3 } else {
4   o[bc] = "y";
5 }
```

where the value of variable `c` is unknown, and the value of variables `ab` and `bc` are a string `a` or `b`, and a string `b` or `c`, respectively. The following abstract object $o^\sharp$ describes all the objects that can be observed at the end of this code fragment:

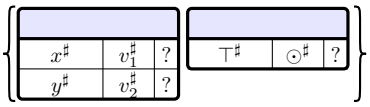
$$(ab^\sharp \mapsto ?, v_1^\sharp) \vee (bc^\sharp \mapsto ?, v_2^\sharp)$$

where $ab^\sharp$ denotes a set of concrete strings $\{\mathtt{a}, \mathtt{b}\}$, $bc^\sharp$ denotes a set of concrete strings $\{\mathtt{b}, \mathtt{c}\}$, $v_1^\sharp$ denotes a concrete string value `"x"`, and $v_2^\sharp$ denotes a concrete string value `"y"`. When we consider a concrete object $o \in \gamma_{\mathbb{O}^\sharp}(o^\sharp)$, the following three properties hold:

- $\mathsf{lookup}(\mathtt{o}, \mathtt{a}) \in \{\mathtt{x}, \odot^\sharp\}$,
- $\mathsf{lookup}(\mathtt{o}, \mathtt{b}) \in \{\mathtt{x}, \mathtt{y}, \odot^\sharp\}$, and
- $\mathsf{lookup}(\mathtt{o}, \mathtt{c}) \in \{\mathtt{y}, \odot^\sharp\}$

where an auxiliary function $\mathsf{lookup} : \mathbb{O} \times \mathbb{V}_{\mathrm{str}} \to \mathbb{V}_{\mathrm{val}}$ that returns a field value designated by a given string in a given object or the `undefined` value when there is no such a field in the object. Since all the abstract fields are annotated with ?, the lookup function may return $\odot^\sharp$. Because two abstract fields are connected by disjunction, the concrete object may have values `x` or `y` in the field `b`.

Example 5.6. The object box notation used in Figure 7 is interpreted as follows:



represents $((x^\sharp \mapsto ?, v_1^\sharp) \wedge (y^\sharp \mapsto ?, v_2^\sharp)) \vee (\top^\sharp \mapsto ?, \odot^\sharp)$.

5.3. String Abstraction

The third layer of the composite abstraction is the string abstraction based on creation sites of string values. Since the set of field names of a JavaScript object is not fixed, it is critical to precisely disambiguate the fields of objects.

We observed that because every field name at run-time is either a string constant or a composition of constants, we can use a simple abstraction to classify strings characterized by their creation sites. Thus, a classified string is either a constant or a set of constants constructed by string operations such as concatenation and replace. The creation-site based string abstraction is parameterized by a basic string abstraction so that a given basic string abstraction is used to abstract the classified strings. While most string abstractions may merge two different string values by a join operation, our

DOI: 10.1002/spe

creation-site string abstraction does not merge two different values when they are distinguished by their creation sites. The basic idea of our creation-site string abstraction is similar to the *allocation-site abstraction* [14, 15], which is commonly used to abstract a set of memory locations in pointer analyses. To represent creation sites of string values, we assume that each instruction in a given program has a unique label $\iota \in \mathbb{L}$. The concrete creation sites $\mathbb{I}$ are tree structure defined as follows:

$$i \in \mathbb{I} ::= \mathtt{L}\,(\iota) \;\mid\; \mathtt{S}\,(\iota, i) \;\mid\; \mathtt{N}\,(\iota, i, i)$$

A creation site of a constant string is a leaf node $\mathtt{L}\,(\iota)$ with its unique label ι as its value. A creation site of a string constructed by a composition operation is a node with one child $\mathtt{S}\,(\iota, i)$ or with two children $\mathtt{N}\,(\iota, i, i)$ denoting a unary or a binary operation, respectively. In both cases, the labels indicate the location at which the operation is performed and the children describe the string operands.

Then, we extend the concrete domain using the creation-site information as follows:

$$\mathbb{V}_e = \mathbb{V}_{\mathrm{val}} \times \mathbb{I} \qquad \mathbb{E}_e = \mathbb{X} \to_{\mathsf{fin}} \mathbb{V}_{\mathrm{val}} \times \mathbb{I} \qquad \mathbb{O}_e = \mathbb{V}_{\mathrm{str}} \to_{\mathsf{fin}} \mathbb{I} \times \mathbb{V}_{\mathrm{e}}$$

Every value is augmented with its creation site. Since a `for-in` loop iterates over a set of fields in an object with their names as index values, we also keep a creation-site information for each field name. Thus, the field value is defined by a product of $\mathbb{I}$ and $\mathbb{V}_e$. Because all the creation sites are kept with values, we do not need to change the semantics of the field lookup and update operations.

Now, we define a creation-site abstraction, which abstracts a set of concrete creation sites characterized by a unique label to an abstract creation site:

Definition 5.6 (Creation-Site Abstraction)**.** An abstract creation site is a unique label $\iota \in \mathbb{L}$.

$$\mathbb{I}^\sharp = \mathbb{L} \uplus \{\top^\sharp, \bot^\sharp\}$$

The concretization of the creation-site abstraction $\gamma_{\mathbb{I}^\sharp}$ is defined as follows:

$$\begin{array}{rcl} \gamma_{\mathbb{I}^\sharp} : \mathbb{I}^\sharp & \longrightarrow & \wp(\mathbb{I}) \\ \iota & \longmapsto & \{\mathtt{L}\,(\iota)\} \cup \{\mathtt{S}\,(\iota, i) \mid i \in \mathbb{I}\} \cup \{\mathtt{N}\,(\iota, i_1, i_2) \mid i_1, i_2 \in \mathbb{I}\} \end{array}$$

The creation-site abstraction ignores the kinds of the concrete creation sites and it abstracts a set of creation sites characterized by its unique label ι to an abstract creation site.

Definition 5.7 (Creation-Site Based String Abstraction)**.** For a given creation-site abstraction $\mathbb{I}^\sharp$ and $\gamma_{\mathbb{I}^\sharp}$, and a string abstraction $\mathbb{V}^\sharp_{\mathrm{str}}$ and γ_{str}, the creation-site based string abstraction is:

$$\mathbb{V}^\sharp_{\mathsf{sa}} = \mathbb{I}^\sharp \to_{\mathsf{fin}} \mathbb{V}^\sharp_{\mathrm{str}}$$

The corresponding concretization is defined as follows:

$$\begin{array}{rcl} \gamma_{\mathsf{sa}} : \mathbb{V}^\sharp_{\mathsf{sa}} & \longrightarrow & \wp(\mathbb{V}_{\mathsf{sa}}) \\ v^\sharp_{\mathsf{sa}} & \longmapsto & \{(i, v) \mid i^\sharp \in \mathbb{I}^\sharp, v^\sharp \in v^\sharp_{\mathsf{sa}}(i^\sharp), i \in \gamma_{\mathbb{I}^\sharp}(i^\sharp), v \in \gamma_{\mathbb{V}^\sharp}(v^\sharp)\} \end{array}$$

Example 5.7. As an example, we use a reduced product of a constant string abstraction and a prefix string abstraction as a basic string abstraction $\mathbb{V}^\sharp_{\mathrm{str}} = \mathbb{V}^\sharp_{\mathsf{const}} \times \mathbb{V}^\sharp_{\mathsf{prefix}}$ and consider the following simple JavaScript code fragment with a loop:

```
1 var i = 10;
2 var v = "x"ι1;
3 while ( --i ) {
4     v = "y" +ι2 i;
5 }
```

When we assume a flow-sensitive analysis of the above code with our creation-site based string abstraction $\mathbb{V}^\sharp_{\mathsf{sa}}$, we can consider the following abstract environment for variable `v` as a result of the analysis at the end of the loop:

$$\mathtt{v} \mapsto \{\iota_1 \mapsto \langle \mathtt{x}, \mathtt{xL}\star \rangle, \iota_2 \mapsto \langle \top_{\mathsf{const}}, \mathtt{yL}\star \rangle\}$$

$$
\begin{array}{lll}
\mathsf{lookup}^\sharp : \mathbb{O}^\sharp \times \mathbb{V}^\sharp_{\mathrm{str}} \to \mathbb{V}^\sharp & & \\
\mathsf{lookup}^\sharp(o^\sharp, k^\sharp) & = \mathsf{lookup}^\sharp_p(o^\sharp, k^\sharp) \sqcup \mathsf{lookup}^\sharp_a(o^\sharp, k^\sharp) & \\
\\
\mathsf{lookup}^\sharp_p(o^\sharp_1 \vee o^\sharp_2, k^\sharp) & = \mathsf{lookup}^\sharp_p(o^\sharp_1, k^\sharp) \sqcup \mathsf{lookup}^\sharp_p(o^\sharp_2, k^\sharp) & \\
\mathsf{lookup}^\sharp_p(c^\sharp, k^\sharp) & = \bigsqcap \{v^\sharp_n \mid \langle k^\sharp_n \mapsto \_, v^\sharp_n \rangle \in c^\sharp, k^\sharp \sqsubseteq k^\sharp_n\} & \text{if } \exists \langle k^\sharp_n \mapsto \_, \_ \rangle \in c^\sharp, k^\sharp \sqsubseteq k^\sharp_n \\
\mathsf{lookup}^\sharp_p(c^\sharp, k^\sharp) & = \bigsqcup \{v^\sharp_n \mid \langle k^\sharp_n \mapsto \_, v^\sharp_n \rangle \in c^\sharp, k^\sharp_n \sqsubseteq k^\sharp\} & \text{otherwise} \\
\\
\mathsf{lookup}^\sharp_a(o^\sharp_1 \vee o^\sharp_2, k^\sharp) & = \mathsf{lookup}^\sharp_a(o^\sharp_1, k^\sharp) \sqcup \mathsf{lookup}^\sharp_a(o^\sharp_2, k^\sharp) & \\
\mathsf{lookup}^\sharp_a(c^\sharp, k^\sharp) & = \odot^\sharp & \text{if } \mathsf{false}^\sharp \sqsubseteq \mathsf{contains}^\sharp(c^\sharp, k^\sharp) \\
\mathsf{lookup}^\sharp_a(c^\sharp, k^\sharp) & = \bot & \text{otherwise} \\
\\
\mathsf{update}^\sharp : \mathbb{O}^\sharp \times \mathbb{V}^\sharp_{\mathrm{str}} \times \mathbb{V}^\sharp \to \mathbb{O}^\sharp & & \\
\mathsf{update}^\sharp(o^\sharp_1 \vee o^\sharp_2, k^\sharp, v^\sharp) & = \mathsf{update}^\sharp(o^\sharp_1, k^\sharp, v^\sharp) \ \vee \ \mathsf{update}^\sharp(o^\sharp_2, k^\sharp, v^\sharp) & \\
\mathsf{update}^\sharp(c^\sharp, k^\sharp, v^\sharp) & = \mathsf{update}^\sharp_c(c^\sharp, k^\sharp, v^\sharp) & \\
\\
\mathsf{contains}^\sharp : \mathbb{O}^\sharp \times \mathbb{V}^\sharp_{\mathrm{str}} \to \mathbb{V}^\sharp_{\mathrm{bool}} & & \\
\mathsf{contains}^\sharp(o^\sharp_1 \vee o^\sharp_2, k^\sharp) & = \mathsf{contains}^\sharp(o^\sharp_1, k^\sharp) \sqcup \mathsf{contains}^\sharp(o^\sharp_2, k^\sharp) & \\
\mathsf{contains}^\sharp(c^\sharp, k^\sharp) & = \mathsf{true}^\sharp & \text{if } \exists \langle k^\sharp_n \mapsto !, \_ \rangle \in c^\sharp,\ k^\sharp_n \sqsubseteq k^\sharp \wedge k^\sharp \sqsubseteq k^\sharp_n \\
\mathsf{contains}^\sharp(c^\sharp, k^\sharp) & = \top^\sharp & \text{if } \exists \langle k^\sharp_n \mapsto \_, \_ \rangle \in c^\sharp,\ k^\sharp_n \sqsubseteq k^\sharp \vee k^\sharp \sqsubseteq k^\sharp_n \\
\mathsf{contains}^\sharp(c^\sharp, k^\sharp) & = \mathsf{false}^\sharp & \text{otherwise}
\end{array}
$$

Figure 10. Functions that manipulate normalized abstract objects

The abstract value of variable `v` is represented by a set of basic abstract string values $\langle$`x`, `xL`$\star\rangle$ and $\langle \top_{\mathsf{const}}$, "`yL`$\star$"$\rangle$, each of which respectively denotes a single constant string "`x`" and a set of strings starting with "`y`". Since each basic abstract string value came from distinct creation sites ι_1 and ι_2, the values are kept as distinct elements instead of being merged as a single abstract value.

In the rest of the paper, $\mathbb{V}^\sharp_{\mathrm{str}}$ denotes the creation-site based string abstraction rather than $\mathbb{V}^\sharp_{\mathsf{sa}}$.

6. ABSTRACT SEMANTICS AND ANALYSIS

We now study the analysis algorithms that infer invariants such as those shown in Figure 7. These rely on the trace partitioning abstraction defined in Section 5.1, on the object abstraction defined in Section 5.2, and on the string abstraction presented in Section 5.3. We first describe the algorithms on the object abstraction that may create partitions and join abstract states while avoiding significant precision losses in Section 6.1. Then, we present the algorithms on the partitioning abstraction that introduce partitions and their joins to achieve a precise analysis of the FCT patterns in Section 6.2.

6.1. Analysis Algorithms Operating on the Object Abstraction

For the simplicity of the algorithms, we maintain the invariant that each abstract object should have in "disjunctive normal form". Thus, each logical formula is a disjunction of "conjunctive abstract objects" that consist of a conjunction of abstract fields or the empty abstract field ϵ. We let $c^\sharp$ denote a conjunctive abstract object. Moreover, when $o^\sharp$ is a disjunctive abstract object, we write $c^\sharp \in o^\sharp$ to denote a conjunctive abstract object $c^\sharp$ that is a sub-term of $o^\sharp$.

Basic object lookup and update operations. We first define abstract counterparts of basic object operations: to lookup a field, to update a field, and to check whether an object contains a field. In this paragraph, we describe functions that do not modify partitions shown in Figure 10, and we describe abstract operations that modify the structure of partitions in the next paragraph.

The abstract lookup operation $\mathsf{lookup}^\sharp$ combines the results of two operations $\mathsf{lookup}^\sharp_p$ and $\mathsf{lookup}^\sharp_a$, which abstract the configurations where the field is present and absent, respectively. Each of these two

DOI: 10.1002/spe

$$
\begin{array}{lll}
\mathsf{plookup}^\sharp : \mathbb{O}^\sharp \times \mathbb{V}^\sharp_{\mathrm{str}} \to \mathcal{P}(\mathbb{V}^\sharp_{\mathrm{str}} \times \mathbb{V}^\sharp) & & \\
\mathsf{plookup}^\sharp(o^\sharp_1 \vee o^\sharp_2, k^\sharp) = \mathsf{plookup}^\sharp(o^\sharp_1, k^\sharp) \cup \mathsf{plookup}^\sharp(o^\sharp_2, k^\sharp) & & \\
\mathsf{plookup}^\sharp(c^\sharp, k^\sharp) & = \{(k^\sharp, \bigsqcap\{v^\sharp_n \,|\langle k^\sharp_n \mapsto \_, v^\sharp_n\rangle \in c^\sharp, k^\sharp \sqsubseteq k^\sharp_n\})\} & \text{if } \exists\langle k^\sharp_n \mapsto \_\rangle \in c^\sharp,\ k^\sharp \sqsubseteq k^\sharp_n \\
\mathsf{plookup}^\sharp(c^\sharp, k^\sharp) & = \{(k^\sharp_n, v^\sharp_i) \,|\langle k^\sharp_n \mapsto \_, v^\sharp_i\rangle \in c^\sharp, k^\sharp_n \sqsubseteq k^\sharp\} & \text{otherwise} \\
\\
\mathsf{alookup}^\sharp : \mathbb{O}^\sharp \times \mathbb{V}^\sharp_{\mathrm{str}} \to \mathcal{P}(\mathbb{V}^\sharp_{\mathrm{str}} \times \mathbb{V}^\sharp) & & \\
\mathsf{alookup}^\sharp(o^\sharp_1 \vee o^\sharp_2, k^\sharp) = \mathsf{alookup}^\sharp(o^\sharp_1, k^\sharp) \cup \mathsf{alookup}^\sharp(o^\sharp_2, k^\sharp) & & \\
\mathsf{alookup}^\sharp(c^\sharp, k^\sharp) & = \{(k^\sharp, \odot^\sharp)\} & \text{if } \mathsf{false}^\sharp \sqsubseteq \mathsf{contains}^\sharp(c^\sharp, k^\sharp) \\
\mathsf{alookup}^\sharp(c^\sharp, k^\sharp) & = \{\} & \text{otherwise} \\
\\
\mathsf{fields}^\sharp : \mathbb{O}^\sharp \to \mathcal{P}(\mathbb{V}^\sharp_{\mathrm{str}}) & & \\
\mathsf{fields}^\sharp(l_1 \vee l_2) & = \mathsf{fields}^\sharp(l_1) \cup \mathsf{fields}^\sharp(l_2) & \\
\mathsf{fields}^\sharp(l_1 \wedge l_2) & = \mathsf{fields}^\sharp(l_1) \cup \mathsf{fields}^\sharp(l_2) & \\
\mathsf{fields}^\sharp(\epsilon) & = \{\} & \\
\mathsf{fields}^\sharp((k^\sharp \mapsto E, v^\sharp)) & = \{k^\sharp\} &
\end{array}
$$

Figure 11. Functions that may create partitions during analysis

functions proceeds by induction over the structure of abstract objects. The abstract update operation $\mathsf{update}^\sharp$ relies on a strong update function $\mathsf{update}^\sharp_c(c^\sharp, k^\sharp, v^\sharp)$, which performs the following two steps:

1. for all $\langle k^\sharp_n \mapsto \_, v^\sharp_i\rangle \in c^\sharp$, if $k^\sharp_n \sqsubseteq k^\sharp$, $c^\sharp[k^\sharp \mapsto (!, v^\sharp)]$; if $k^\sharp \sqsubseteq k^\sharp_n$, $c^\sharp[k^\sharp_n \mapsto (?, v^\sharp \sqcup v^\sharp_i)$; otherwise, $c^\sharp$.
2. $c^\sharp[k^\sharp \mapsto (!, v^\sharp)]$.

Last, $\mathsf{contains}^\sharp$ provides an abstract test for the existence of a field described by an abstract string.

Operations that may create partitions Figure 11 shows the semantic functions that may create partitions when analyzing general loops and `for-in` loops:

- $\mathsf{plookup}^\sharp$ creates a partition that corresponds to the case where a field is present. The call $\mathsf{plookup}^\sharp(o^\sharp, k^\sharp)$ returns a set of pairs made of a field name and a value in $o^\sharp$, where each pair is designated by $k^\sharp$.
- $\mathsf{alookup}^\sharp$ creates a partition corresponding to the case where a field is absent. The call $\mathsf{alookup}^\sharp(o^\sharp, k^\sharp)$ returns the singleton pair of a field name and the `undefined` value when there is an absent field designated by $k^\sharp$ in $o^\sharp$. Otherwise, it returns the empty set.
- $\mathsf{fields}^\sharp$ returns a set of the fields of an object. The call $\mathsf{fields}^\sharp(o^\sharp)$ returns a set of field names of $o^\sharp$ that are to be iterated by `for-in` loops.

The functions $\mathsf{plookup}^\sharp$ and $\mathsf{alookup}^\sharp$ generate partitions for general loops at a field lookup instruction. The function $\mathsf{fields}^\sharp$ plays the same role for `for-in` loops.

The following lemmas prove the soundness of the functions $\mathsf{plookup}^\sharp$, $\mathsf{alookup}^\sharp$, and $\mathsf{fields}^\sharp$.

Lemma 6.1 (Soundness of the lookup functions)**.** Given an abstract object $o^\sharp$ and an abstract string value $k^\sharp$, the union of two lookup functions computes an over-approximation of the concrete counter part $\mathsf{lookup} : \mathbb{O} \times \mathbb{V}_{\mathrm{str}} \to \mathbb{V}_{\mathrm{val}}$:

$$
\begin{array}{l}
\forall o \in \gamma_{\mathbb{O}^\sharp}(o^\sharp),\ \forall k \in \gamma_{\mathrm{str}}(k^\sharp),\ \forall v \in \mathbb{V}_{\mathrm{val}}, \\
v \in \mathsf{lookup}(o, k) \Rightarrow \exists(k^\sharp_n, v^\sharp_n) \in V,\ k \in \gamma_{\mathrm{str}}(k^\sharp_n) \wedge v \in \gamma_{\mathbb{V}^\sharp}(v^\sharp_n)
\end{array}
$$

where $V = \mathsf{plookup}^\sharp(o^\sharp, k^\sharp) \cup \mathsf{alookup}^\sharp(o^\sharp, k^\sharp)$.

Lemma 6.2 (Soundness of the field collection function)**.** Given an abstract object $o^\sharp$, the function $\mathsf{fields}^\sharp$ computes an over-approximation of the concrete counter part $\mathsf{fields} : \mathbb{O} \to \mathbb{V}_{\mathrm{val}}$:

$$
\forall o \in \gamma_{\mathbb{O}^\sharp}(o^\sharp),\ \exists k^\sharp \in \mathsf{fields}^\sharp(o^\sharp),\ \mathsf{fields}(o) \in \gamma_{\mathrm{str}}(k^\sharp).
$$

Join algorithm. We now discuss the abstract join algorithm that over-approximates the concrete union of the concretizations of two abstract objects. Such an over-approximation is not unique, and there exists generally no best solution, thus this algorithm aims at avoiding significant precision losses. For instance, in Figure 7, the analysis computes $\circ 2$ at point ⓘ by joining the three objects $\circ 2$ of ⓒ.

The join algorithm takes two abstract objects in "disjunctive normal form" and computes an over-approximation of all the concrete objects they represent. Given two such abstract objects $\circ 0$ and $\circ 1$, the join algorithm first searches for a matching between the conjunctive components of $\circ 0$ and of $\circ 1$, that can be merged without a significant precision loss. Such pairs of components are characterized by a *join condition* that we discuss below. When two conjunctive abstract objects $c_0^\sharp$ and $c_1^\sharp$ satisfy this condition and can be joined, they are replaced with a new conjunctive abstract object, where each abstract field approximates abstract fields of $c_0^\sharp$ and of $c_1^\sharp$ as follows:

$$c_0^\sharp \sqcup c_1^\sharp = \{k^\sharp \mapsto (E_1 \sqcup E_2, v_1 \sqcup v_2) \mid \begin{array}{l} k^\sharp \in \mathsf{fields}^\sharp(c_0^\sharp) \cup \mathsf{fields}^\sharp(c_1^\sharp), \\ v_1^\sharp = \mathsf{lookup}^\sharp(c_0^\sharp, k^\sharp),\ v_2^\sharp = \mathsf{lookup}^\sharp(c_1^\sharp, k^\sharp), \\ (E_1, \_) = c_0^\sharp(k^\sharp),\ (E_2, \_) = c_1^\sharp(k^\sharp)\} \end{array}$$

where the $\mathsf{fields}^\sharp$ function collects all the fields in a given object, and the join of two predicates $E_1 \sqcup E_2$ is ! when both E_1 and E_2 are !, and ? for all the other cases. When $c_0^\sharp$ and $c_1^\sharp$ do not satisfy the condition, the join algorithm returns a disjunction.

The join algorithm decides whether to merge two abstract object fragments using the join condition. Let us consider the join of a pair of abstract fields. When the name of either of these fields subsumes the name of the other that is different from itself, their names and values can be merged conservatively but the resulting abstract field may fail to capture an optimal field correspondence relation. For example, the result of joining two abstract fields $(x^\sharp, !, v_1^\sharp)$ and $(\top^\sharp, ?, v_2^\sharp)$ is $(\top^\sharp, ?, \{v_1^\sharp, v_2^\sharp\})$ because the name $\top^\sharp$ subsumes the name of the other field $x^\sharp$. Thus, we avoid joining abstract objects containing such pairs of abstract fields as characterized by the definition below:

Definition 6.1 (Join Condition)**.** Two sets of abstract fields $P_1^\sharp$ and $P_2^\sharp$ are *correlated* if and only if $\forall k_1^\sharp \in N(P_1^\sharp),\ \forall k_2^\sharp \in N(P_2^\sharp),\ \gamma_{\mathrm{str}}(k_1^\sharp) \cap \gamma_{\mathrm{str}}(k_2^\sharp) \neq \emptyset$ where $N(P^\sharp)$ (resp., $V(P^\sharp)$) denotes the names (resp., values) of the fields in $P^\sharp$.

Two abstract objects $c_1^\sharp$ and $c_2^\sharp$ satisfy the *join condition* if and only if, for all the correlated abstract fields $P_1^\sharp$ and $P_2^\sharp$ in $c_1^\sharp$ and $c_2^\sharp$, either of the following three conditions holds:

- $N_1^\sharp = N_2^\sharp$
- $N_1^\sharp \subseteq N_2^\sharp \wedge V_1^\sharp \sqsubseteq V_2^\sharp$, or, symmetrically $N_2^\sharp \subseteq N_1^\sharp \wedge V_2^\sharp \sqsubseteq V_1^\sharp$
- $N_1^\sharp \cap N_2^\sharp \neq \emptyset \wedge V_1^\sharp = V_2^\sharp$

where $N_1^\sharp = \{\gamma_{\mathrm{str}}(k^\sharp) \mid k^\sharp \in N(P_1^\sharp)\},\ N_2^\sharp = \{\gamma_{\mathrm{str}}(k^\sharp) \mid k^\sharp \in N(P_2^\sharp)\},\ V_1^\sharp = \cup\, V(P_1^\sharp)$, and $V_2^\sharp = \cup\, V(P_2^\sharp)$.

We write $o_1^\sharp \simeq o_2^\sharp$ to denote the fact that the abstract objects $o_1^\sharp$ and $o_2^\sharp$ satisfy the join condition. For example, $\circ 2$ in the third disjunct at ⓒ does not satisfy the join condition with $\circ 2$ in either of the other two disjuncts at that point. On the other hand, the abstract objects corresponding to $\circ 2$ in the first two disjuncts at ⓒ satisfy this condition, and will be joined together.

Figure 12 shows the algorithm of the join operator $\mathsf{join}^\sharp$ defined by the join algorithm and the join condition.

Theorem 6.1 (Soundness of the join algorithm)**.** Given two abstract objects $o_0^\sharp$ and $o_1^\sharp$, the abstract join computes an over-approximation of the union of the two abstract objects:

$$\forall i \in \{0, 1\},\ \forall o \in \mathbb{O}, o \in \gamma_{\mathbb{O}^\sharp}(o_i^\sharp) \Rightarrow o \in \gamma_{\mathbb{O}^\sharp}(\mathsf{join}^\sharp(o_0^\sharp, o_1^\sharp)).$$

Widening algorithm. In addition to over-approximation of concrete unions like $\mathsf{join}^\sharp$, widening also ensures termination of abstract iterates. It relies on an algorithm that is similar to that of $\mathsf{join}^\sharp$ except that it applies a widening $\triangledown : \mathbb{V}^\sharp \times \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ operation when a pair of abstract fields should

DOI: 10.1002/spe

function $\mathsf{join}^\sharp(o_0^\sharp, o_1^\sharp)\colon \mathbb{O}^\sharp \times \mathbb{O}^\sharp \to \mathbb{O}^\sharp$
  $c_{new}^\sharp \leftarrow \epsilon$; $o_{need}^\sharp \leftarrow o_1^\sharp$
  for all $c_0^\sharp \in o_0^\sharp$ **do**
    $c^\sharp \leftarrow$ `null`
    for all $c_1^\sharp \in o_{need}^\sharp$ **do**
      if $c_0^\sharp \simeq c_1^\sharp$ **then** $c^\sharp \leftarrow c_1^\sharp$; $o_{need}^\sharp \leftarrow o_{need}^\sharp \setminus \{c_1^\sharp\}$; **break** **end if**
    end for
    if $c^\sharp \neq$ `null` **then** $c_{new}^\sharp \leftarrow c_{new}^\sharp \vee (c_0^\sharp \sqcup c^\sharp)$ **else** $c_{new}^\sharp \leftarrow c_{new}^\sharp \vee c_0^\sharp$ **end if**
  end for
  return $c_{new}^\sharp \vee o_{need}^\sharp$
end function

Figure 12. Algorithm of the $\mathsf{join}^\sharp$ operator

get merged as they satisfy the join condition (Definition 6.1). Therefore, the widening of $c_0^\sharp$ and $c_1^\sharp$ is defined as follows:

$$c_0^\sharp \triangledown c_1^\sharp = \{k^\sharp \mapsto (E_1 \sqcup E_2, v_1 \triangledown v_2) \mid \begin{array}{l} k^\sharp \in \mathsf{fields}^\sharp(c_0^\sharp) \cup \mathsf{fields}^\sharp(c_1^\sharp), \\ v_1^\sharp = \mathsf{lookup}^\sharp(c_0^\sharp, k^\sharp),\ v_2^\sharp = \mathsf{lookup}^\sharp(c_1^\sharp, k^\sharp), \\ (E_1, \_) = c_0^\sharp(k^\sharp),\ (E_2, \_) = c_1^\sharp(k^\sharp)\} \end{array}$$

where the $\mathsf{fields}^\sharp$ function collects all the fields in a given object, and the join of two predicates $E_1 \sqcup E_2$ is ! when both E_1 and E_2 are !, and ? for all the other cases.

The widening algorithm ensures the termination of abstract iterates when the height of the baseline string abstraction is finite because the second join condition ($N_1^\sharp \subseteq N_2^\sharp$ or $N_2^\sharp \subseteq N_1^\sharp$) is satisfied for at most finitely many iterations.

Theorem 6.2 (Widening algorithm termination)**.** The widening algorithm computes an over-approximation of its arguments and ensure the termination of abstract iterates.

6.2. Analysis Algorithms Operating on the Partitioning Abstraction

We now present the static analysis algorithms that create partitions dynamically, as we have shown in Section 3. The functions that we define below rely on the operations on abstract objects presented in Section 6.1.

Our analysis is based on a standard flow-sensitive abstract interpretation approach, which utilizes the partitioning abstraction $\mathbb{D}^\sharp = \mathbb{T}^\sharp \to \mathbb{M}^\sharp$ defined in Section 5.1. In this setup, we let the abstract semantics of an instruction i be a function $[\![i]\!]_I^\sharp : (\mathbb{T}^\sharp \times \mathbb{M}^\sharp) \to \mathcal{P}(\mathbb{T}^\sharp \times \mathbb{M}^\sharp)$ that takes an abstract pre-condition as an input and returns a conservative abstract post-condition. Since new partitions may be created at any point during the analysis, this abstract semantic function returns a set of abstract memories paired with their corresponding abstract trace elements.

In the following, we discuss the analysis of loops that contain plural FCT patterns. Note that the analysis algorithm for general loops that contain plural FCT patterns is also applicable to loop-free code fragments that contain plural FCT patterns.

Identifying key variables by a syntactic pre-analysis. A precise analysis of loops requires an accurate knowledge of key variables. In order to identify key variables of FCT patterns, we use a syntactic pre-analysis, which identifies variables that are used as index variables in field lookup and update instructions in the loop body. When the pre-analysis identifies `x` as a key variable of an FCT pattern, it adds the `merge(x)` instruction at the end of the loop. For example, the pre-analysis result of the `while` loop in Figure 6(a) is shown in Figure 14(a). Because `v` appears in both a field lookup and a field update in the loop body, it is identified as a key variable. On the contrary, because `i` is used only for a field lookup, it is not a key variable.

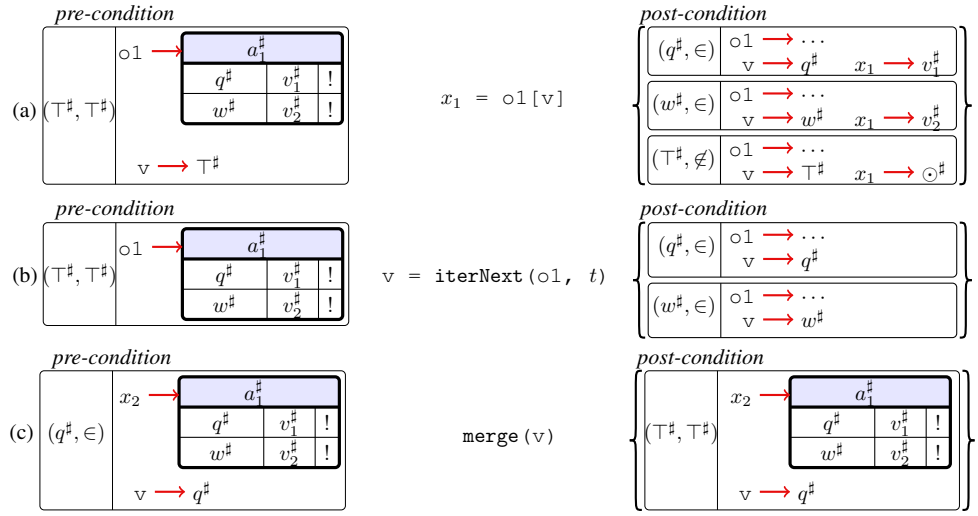


Figure 13. Example cases for (a) the field lookup instruction with the key variable `v`, (b) the `iterNext` instruction, and (c) the `merge` instruction by the key variable `v` with the first post-state of both (a) and (b)

```
var i = arr.length;
while ( i-- ) {
   v = arr[i];
   o2[v] = o1[v];
   merge(v);
 }
```

(a)

```
t = iterInit(o2);
b = iterHasNext(t);
while (b) {
  v = iterNext(o2, t);
  o1[v] = o2[v];
  b = iterHasNext(t);
  merge(v);
}
```

(b)

Figure 14. Desugared versions of `while` and `for-in` loops shown in Figure 6

Analysis of general loops. Using the pre-analysis result, the analysis creates partitions when it encounters a field lookup indexed by a key variable.

It essentially creates one partition per possible (defined or undefined) abstract value produced as the result of the lookup operation. For example, Figure 13(a) shows such cases when the field lookup may return either the field $\{(q^\sharp, v_1^\sharp)\}$ or the field $\{(w^\sharp, v_2^\sharp)\}$, or the `undefined` value $\{\odot^\sharp\}$ (if the field is absent in the object). The variable x_1 is temporarily created to hold the result of the lookup. Each generated partition corresponds to a field correspondence relation. The partitions are merged at the end of the loop body by the `merge` instruction as shown in Figure 13(c). The abstract semantics of the field lookup instruction $x_1 = x_2\,[\,x_3\,]$ is as follows:

$$
\begin{aligned}
&[\![x_1 = x_2\,[\,x_3\,]_l]\!]^\sharp_I(t^\sharp, m^\sharp) = \\
&\quad \{(((l, x_3, x_2, \in), (l, x_3, k^\sharp)), m_0^\sharp) \mid \exists a^\sharp \in m^\sharp(x_2),\ \exists (k^\sharp, v^\sharp) \in \mathsf{plookup}^\sharp(m^\sharp(a^\sharp), m^\sharp(x_3)), \\
&\qquad\qquad m_0^\sharp = m^\sharp[x_1 \mapsto v^\sharp][x_3 \mapsto k^\sharp]\} \\
&\quad \cup \{(((l, x_3, x_2, \notin), (l, x_3, k^\sharp)), m_0^\sharp) \mid \exists a^\sharp \in m^\sharp(x_2),\ \exists (k^\sharp, v^\sharp) \in \mathsf{alookup}^\sharp(m^\sharp(a^\sharp), m^\sharp(x_3)), \\
&\qquad\qquad m_0^\sharp = m^\sharp[x_1 \mapsto v^\sharp]\}
\end{aligned}
$$

where l denotes the syntactic program point of the instruction.

Analysis of for-in loops. The desugared version of the `for-in` loop in Figure 6(b) is shown in Figure 14(b). The semantics of the `for-in` statement is as follows: it iterates over the fields of a given object `o1`, and lets the index variable `v` range over their names. To mimic this behavior, the instruction `v = iterInit(o1)` generates the list of the field names of object `o1`; then, `iterHasNext(t)` checks whether there exists a field that has not been visited yet, and finally, `iterNext(o1, t)`

DOI: 10.1002/spe

returns the next field name to visit. Since the order that a `for-in` loop visits fields is undefined in the language semantics [12], we simply assume a non-deterministic visiting order. We add `merge(v)` at the end of each loop to denote the end of the FCT pattern with the key variable `v`. In order to reason over distinct fields separately, the analysis performs dynamic partitioning during the analysis of the instruction `v = iterNext(o1, t)`, and generates partitions for each possible field name. Because the analysis needs to analyze field names in each partition precisely so that it can analyze the field lookup instruction at line 5 precisely, it collects a set of field names in the given object `o1`, and generates partitions for each field name in this set. In Figure 13(b), we illustrate such a case. The partitions are merged at the end of the loop body, in the same way as for the `while` loop case. The abstract semantics of the `iterNext` and `merge` instructions are as follows:

$$[\![x_1 = \texttt{iterNext}\,(x_2, x_3)_{\,l}]\!]^\sharp_I(t^\sharp, m^\sharp) = \\ \{(((l, x_1, x_2, \in), (l, x_1, k^\sharp)), m_0^\sharp) \mid \exists a^\sharp \in m^\sharp(x_2),\ \exists k^\sharp \in \mathsf{fields}^\sharp(m^\sharp(a^\sharp)),\ m_0^\sharp = m^\sharp[x_1 \mapsto k^\sharp]\}$$

$$[\![\texttt{merge}\,(\texttt{v})_{\,l}]\!]^\sharp_I(((\_, \texttt{v}, \_, \_), (\_, \texttt{v}, \_)), m^\sharp) = \{((\top^\sharp, \top^\sharp), m^\sharp)\}$$
$$[\![\texttt{merge}\,(\texttt{v})_{\,l}]\!]^\sharp_I(t^\sharp, m^\sharp) = \{(t^\sharp, m^\sharp)\} \qquad \text{otherwise}$$

where l denotes the syntactic program point of the instruction.

Analysis of loops. We now define the abstract transfer function for loops. The abstract semantics of the `while` instruction is defined as follows:

$$[\![\texttt{while}\,(e)\quad i]\!]^\sharp_I(t^\sharp, m^\sharp) = \mathrm{fix}^\sharp_{(t^\sharp, m^\sharp)} \lambda(t_a^\sharp, m_a^\sharp).(t_a^\sharp, m_a^\sharp) \sqcup [\![i]\!]^\sharp_P(t_a^\sharp, m_a^\sharp)$$

where the abstract transfer function for partitioning $[\![i]\!]^\sharp_P$ is defined by:

$$[\![i]\!]^\sharp_P : (\mathbb{T}^\sharp \times \mathbb{M}^\sharp) \longrightarrow (\mathbb{T}^\sharp \times \mathbb{M}^\sharp)$$
$$(t_a^\sharp, m_a^\sharp) \longmapsto (t_0^\sharp, \sqcup\{m^\sharp \mid (t^\sharp, m^\sharp) \in S^\sharp, t^\sharp \sqsubseteq t_0^\sharp\}) \text{ where } \begin{cases} S^\sharp &= [\![i]\!]^\sharp_I(t_a^\sharp, m_a^\sharp) \\ t_0^\sharp &= \mathsf{Ctx}(S^\sharp) \end{cases}$$
$$\mathsf{Ctx} : \mathcal{P}(\mathbb{T}^\sharp \times \mathbb{M}^\sharp) \longrightarrow \mathbb{T}^\sharp$$
$$S^\sharp \longmapsto t^\sharp \quad \text{such that } \forall t^\sharp \in \mathbb{T}^\sharp,\ \forall (t_0^\sharp, \_) \in S^\sharp,\ (t^\sharp, \_) \in S^\sharp \wedge t_0^\sharp \sqsubseteq t^\sharp.$$

Since the partitions generated in the loop body are merged at the end of the loop by the `merge` instruction, Ctx can find one trace abstraction $t^\sharp$ that satisfies the condition $\forall (t_0^\sharp, m_0^\sharp) \in S^\sharp,\ t_0^\sharp \sqsubseteq t^\sharp$. The $\mathrm{fix}^\sharp_{(t^\sharp, m^\sharp)}$ operator computes a sequence of the abstract iterates, and guarantees termination by a widening operator using the the abstract object widening defined in Section 6.1.

The full abstract transfer functions $[\![i]\!]^\sharp_I : (\mathbb{T}^\sharp \times \mathbb{M}^\sharp) \to \mathcal{P}(\mathbb{T}^\sharp \times \mathbb{M}^\sharp)$ and the abstract semantic functions for expressions $[\![e]\!]^\sharp_E : \mathbb{M}^\sharp \to \mathbb{V}^\sharp$ are shown in Figure 15. This analysis terminates and computes a sound over-approximation for the set of concrete program behaviors as shown in the following theorems.

Theorem 6.3 (Soundness of transfer functions)**.** The abstract transfer functions are sound as the following property holds for any instruction i:

$$\forall t^\sharp \in \mathbb{T}^\sharp,\ \forall m^\sharp \in \mathbb{M}^\sharp,\ \forall m \in \gamma_{\mathbb{M}^\sharp}(m^\sharp),\ \exists (\_, m_0^\sharp) \in [\![i]\!]^\sharp_I(t^\sharp, m^\sharp),\ \tau[i](m) \in \gamma_{\mathbb{M}^\sharp}(m_0^\sharp).$$

Theorem 6.4 (Analysis soundness and termination)**.** Given program i and an abstract initial state $m^\sharp$, the computation of $[\![i]\!]^\sharp_P(\top^\sharp, m^\sharp)$ terminates, and the derived post-condition is sound:

$$[\![i]\!]_C(\gamma(\top^\sharp, m^\sharp)) \subseteq \gamma([\![i]\!]^\sharp_P(\top^\sharp, m^\sharp))$$

where $\gamma(t^\sharp, m^\sharp) = \gamma_{\mathbb{M}^\sharp}(m^\sharp)$.

$$[\![x = e_l]\!]^\sharp_I(t^\sharp, m^\sharp) \;= \{(t^\sharp, m^\sharp[x \mapsto [\![e]\!]^\sharp_E(m^\sharp)])\}$$

$$[\![x = \{\}_l]\!]^\sharp_I(t^\sharp, m^\sharp) = \{(t^\sharp, m^\sharp[a^\sharp \mapsto \epsilon^\sharp][x \mapsto a^\sharp])\} \quad \text{such that } a \in \gamma_{\text{str}}(a^\sharp) \text{ with a new address } a$$

$$\begin{aligned}[\![x_1 = x_2\,[\,x_3\,]_l]\!]^\sharp_I(t^\sharp, m^\sharp) = \\ \{((\!(l, x_3, x_2, \in), (l, x_3, k^\sharp)\!), m_0^\sharp) \mid \exists a^\sharp \in m^\sharp(x_2),\ \exists (k^\sharp, v^\sharp) \in \mathsf{plookup}^\sharp(m^\sharp(a^\sharp), m^\sharp(x_3)),\\ m_0^\sharp = m^\sharp[x_1 \mapsto v^\sharp][x_3 \mapsto k^\sharp]\} \\ \cup \{((\!(l, x_3, x_2, \notin), (l, x_3, k^\sharp)\!), m_0^\sharp) \mid \exists a^\sharp \in m^\sharp(x_2),\ \exists (k^\sharp, v^\sharp) \in \mathsf{alookup}^\sharp(m^\sharp(a^\sharp), m^\sharp(x_3)),\\ m_0^\sharp = m^\sharp[x_1 \mapsto v^\sharp]\}\end{aligned}$$

$$[\![x_1\,[\,x_2\,] = x_3{}_l]\!]^\sharp_I(t^\sharp, m^\sharp) \qquad = \{(t^\sharp, \textstyle\bigsqcup_{a^\sharp \in m^\sharp(x_1)} m^\sharp[a^\sharp \mapsto \mathsf{update}^\sharp(m^\sharp(a^\sharp), m^\sharp(x_2), m^\sharp(x_3))])\}$$

$$[\![x_1 = \texttt{iterInit}\,(x_2)_{\,l}]\!]^\sharp_I(t^\sharp, m^\sharp) = \{(t^\sharp, m^\sharp)\}$$

$$\begin{aligned}[\![x_1 = \texttt{iterNext}\,(x_2\texttt{,}\,x_3)_{\,l}]\!]^\sharp_I(t^\sharp, m^\sharp) = \\ \{((\!(l, x_1, x_2, \in), (l, x_1, k^\sharp)\!), m_0^\sharp) \mid \exists a^\sharp \in m^\sharp(x_2), \exists k^\sharp \in \mathsf{fields}^\sharp(m^\sharp(a^\sharp)),\ m_0^\sharp = m^\sharp[x_1 \mapsto k^\sharp]\}\end{aligned}$$

$$[\![x_1 = \texttt{iterHasNext}\,(x_2)_{\,l}]\!]^\sharp_I(t^\sharp, m^\sharp) \quad = \{t^\sharp, m^\sharp[x_1 \mapsto \top^\sharp_{\mathsf{str}}]\}$$

$$[\![\texttt{merge}\,(\texttt{v})_{\,l}]\!]^\sharp_I(((\_, \texttt{v}, \_, \_), (\_, \texttt{v}, \_)), m^\sharp) = \{((\top^\sharp, \top^\sharp), m^\sharp)\}$$

$$[\![\texttt{merge}\,(\texttt{v})_{\,l}]\!]^\sharp_I(t^\sharp, m^\sharp) = \{(t^\sharp, m^\sharp)\} \qquad \text{otherwise}$$

$$[\![\texttt{while}\,(e)\quad i]\!]^\sharp_I(t^\sharp, m^\sharp) = \text{fix}^\sharp_{(t^\sharp, m^\sharp)} \lambda(t_a^\sharp, m_a^\sharp).(t_a^\sharp, m_a^\sharp) \sqcup [\![i]\!]^\sharp_P(t_a^\sharp, m_a^\sharp)$$

$$[\![i_1\,\texttt{;}\,i_2]\!]^\sharp_I(t^\sharp, m^\sharp) \qquad = \{(t_2^\sharp, m_2^\sharp) \,|\, (t_1^\sharp, m_1^\sharp) \in [\![i_1]\!]^\sharp_I(t^\sharp, m^\sharp), (t_2^\sharp, m_2^\sharp) \in [\![i_2]\!]^\sharp_I(t_1^\sharp, m_1^\sharp)\}$$

$$[\![k]\!]^\sharp_E(m^\sharp) \qquad = k^\sharp \qquad \text{such that } k \in \gamma_{\text{str}}(k^\sharp)$$

$$[\![x]\!]^\sharp_E(m^\sharp) \qquad = m^\sharp(x)$$

$$[\![e_1 + e_2]\!]^\sharp_E(m^\sharp) = [\![e_1]\!]^\sharp_E(m^\sharp) \;\cdot^\sharp\; [\![e_2]\!]^\sharp_E(m^\sharp)$$

where $\epsilon^\sharp \in \mathbb{O}^\sharp$ denotes an abstract empty object and $\cdot^\sharp : (\mathbb{V}^\sharp_{\text{str}} \times \mathbb{V}^\sharp_{\text{str}}) \to \mathbb{V}^\sharp_{\text{str}}$ is an abstract string concatenation function.

Figure 15. The full abstract transfer functions $[\![i]\!]^\sharp_I$ and the abstract semantic functions for expressions $[\![e]\!]^\sharp_E$

7. OPTIMIZING THE ANALYSIS BY AVOIDING RE-COMPUTATIONS

In this section, we discuss an important implementation technique, that is required to ensure the scalability of a static analysis based on our composite abstraction. As the analysis of an FCT pattern requires the use of partitioning technique, it also naturally induces a duplication of computations. In particular, when the analysis of an FCT pattern needs to resolve the effect of a function call for a partition, this results in the composite abstraction distinguishing more calling contexts, which in turns results in the analysis of the function body being repeated many times. However, when these function calls have the same effective arguments, these re-computations are unnecessary. This phenomenon poses a threat to the scalability of static analyses based on composite abstraction. To avoid this potential analysis cost issue, it is important to mitigate the cost of such re-computations. In the following of this section, we present a technique that effectively eliminates duplicated function body analyses in a context sensitive analysis based on composite abstraction.

A Simple FCT Pattern Inducing Function Re-Analysis. We consider the simple JavaScript program in Figure 16. The code contains an FCT pattern of $\texttt{o1[}f(\texttt{v})\texttt{]} = g(\texttt{o2[v]})$ at lines 7–9 where variable `o1`, `o2`, `v` are the corresponding variables in the FCT pattern, $f(\texttt{v})$ on the left-hand

DOI: 10.1002/spe

```
1  var h = "foo", o1 = { }, o2 = { x: 1, y: 2 };
2  function k() {
3    return h;
4  }
5
6  for (var v in o2) {
7    t0 = o2[v];
8    t1 = k();
9    o1[v] = t0 + t1;
10 }
```

Figure 16. A simple FCT pattern that contains a function call.

```
2  function k() {
```

$(8, \texttt{x}) : \{\texttt{h} \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"x"}, \texttt{t0} \mapsto 1\}$
$(8, \texttt{y}) : \{\texttt{h} \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"y"}, \texttt{t0} \mapsto 2\}$

```
3    return h;
```

$(8, \texttt{x}) : \{rtn \mapsto \texttt{"foo"}, \texttt{h} \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"x"}, \texttt{t0} \mapsto 1\}$
$(8, \texttt{y}) : \{rtn \mapsto \texttt{"foo"}, \texttt{h} \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"y"}, \texttt{t0} \mapsto 2\}$

```
4  }
```

Figure 17. Input/output states for function k caused by two partitions "x" and "y" with its calling context 8.

side corresponds to the identity function, and $g(\texttt{o2[v]})$ on the right-hand side corresponds to the concatenation of two values produced by the lookup operation at line 7 and the function call at line 8.

The analysis based on composite abstraction partitions the analysis of the loop body, so as to reason separately over distinct occurrences of field lookup operations in the FCT pattern. Failure to distinguish the field lookup operations would cause the loss of field correspondence relations. The analysis based on composite abstraction performs dynamic partitioning during the analysis of the FCT pattern at lines 7–9, hereby distinguishing calling contexts. We let the line numbers that correspond to call-sites designate the calling contexts. The analysis generates two partitions named x and y for the lookup operation at line 7. In each partition, the values of variables v and t0 are "x" and 1, and "y" and 2, respectively. Since the analysis merges the partitions only at the end of the loop body, the function call at line 8 is computed twice, namely once for each partition.

We describe input/output analysis states for function k in Figure 17. Each input or output state is labeled by a pair that consists of the call site line number, and the partition name. For example, there are two input states distinguished by a pair of the calling context 8 and partitions x and y as in $(8, \texttt{x})$ and $(8, \texttt{y})$. The call site name 8 designates all the executions that reach the function from a call at line 8. In each partition, the value of variable h is "foo" and the values of variable v are "x" and "y", respectively.

Analyzing the body of function k once per partition is unnecessary here, since both calls produce the same information. Indeed, the body of the function only looks up the value of variable h, and returns this value (we note rtn the return value in the output state). Therefore analyzing the body of this function twice will produce twice the very same results. In larger codes, a similar effect would likely cause even larger numbers of useless function body re-analyses, and be detrimental to the overall analysis scalability as the number of times the body of the function is analyzed is multiplied by the number of calling contexts and partitions in the FCT pattern where the call occurs.

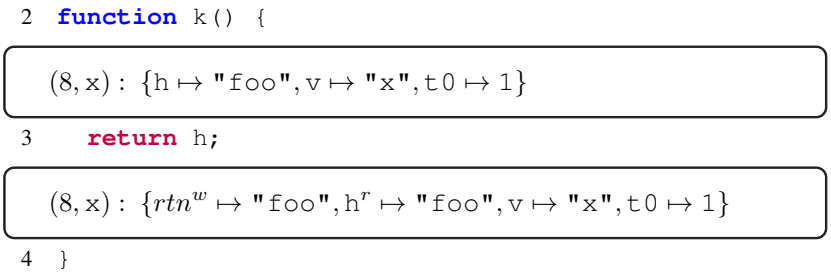


Figure 18. The analysis of function k with extra annotations about write (^{w}) and read (^{r}) information during the analysis.

However, we remark that reusing analysis results obtained for the first call to function k for subsequent calls with similar call state is not straightforward in general. Indeed, in this example, the values of variables v and t0 are different for the two calls to function k, thus standard pre-condition tabulation techniques would require the body of the function be re-analyzed. However, these variables are not relevant to the body of function k, thus analyzing the function body only once appears as a viable approach in this case.

An Approach based on Conditional Summaries. To prevent the analysis from carrying out redundant function body analyses, we propose to utilize localized function summaries. Our approach is based on the so-called *localization* technique introduced in [16, 17, 18]. The key idea is to compute function summaries, that are conditioned by the semantic properties which are relevant to the analysis of the function body. The conditional summary provides a way to reuse an existing analysis result, instead of re-analyzing the function body, which would be a duplicate computation. On the other hand, when the analysis encounters a call site, and when the existing conditioned summary does not match the semantic information computed for the new calling context: then, the summary is not used and the function body is analyzed again, as expected (note that this function body analysis is not a duplicate computation). Our technique guarantees that the output state computed by the summary is equal to the output state produced by the analysis of the function if an input state satisfies the condition of the summary.

Example. Before we formalize conditional summaries, we demonstrate the underlying intuition using the analysis of function k in Figure 16 as an example. As we have shown in Figure 17, a standard analysis algorithm applied to the FCT pattern at lines 7–9 needs to analyze the body of function k twice. Furthermore, we observed that the second analysis of the body of k is redundant. The goal of the conditioned summaries is to safely avoid this recomputation.

To define the analysis based on conditional summaries, we augment abstract states with information about memory locations that are read or written to in the body of functions. More precisely, we let the analysis collect information about which memory locations of the input state may be read in the function body, and which memory locations of the output state may be written by the function body. We present the results of this novel extended analysis of function k, for the context $(8, \mathtt{x})$ in Figure 18. We annotated with a superscript ^{r} the variables that are marked as possibly read, and with a superscript ^{w} the variables that may be written, in the abstract states.

With the annotated output state, we construct a conditional summary of the function as a following pair:

$$(\{\mathtt{h} \mapsto \texttt{"foo"}\}, \{rtn \mapsto \texttt{"foo"}\}).$$

The first and second element of the pair summarize the read and write of the analysis of the function body. The first element of the pair denotes the information that are read during the analysis of the function body, variables of which were annotated with a superscript ^{r} in the augmented output state

DOI: 10.1002/spe

and values of each variable came from the corresponding variables in the input state of the function. Since the analysis of function `k` reads variable `h`, the first element of the pair consists of variable `h` and its value `"foo"`. Likewise, the second element of the pair denotes the information that are written during the analysis, variables of which were annotated with a superscript ^{w} in the augmented output state and values of each variable came from the corresponding variables in the output state of the function. Since the analysis of function `k` writes into variable rtn, the second element of the pair consits of variable rtn and its value. We respectively call the first and second element of the pair the *summary condition* and the *summary output* of the summary.

Since all the values in the summary condition equals to the corresponding values in the input state of function `k` for the context $(8, \mathtt{y})$, we instantiate the analysis result from the summary. The value of variable `h` in the summary condition is `"foo"`, which means that the analysis read the variable `h` and its value was `"foo"` during the course of the analysis. Since the value of variable `h` of the input state for the context $(8, \mathtt{y})$ is `"foo"`, we confirm that the analysis with the input state for the context $(8, \mathtt{y})$ compute the same information as the summary output, and thus we can use the summary to compute the analysis result of the input state.

To instantiate the summary, we overwrite the summary output on top of the input state. The summary output provides the information that is written during the analysis of the function. Thus, an overwrite of the summary output on the input state generates the output state of the analysis of the function as follows:

$$\{rtn \mapsto \mathtt{"foo"}, \mathtt{h} \mapsto \mathtt{"foo"}, \mathtt{v} \mapsto \mathtt{"y"}, \mathtt{t0} \mapsto 2\}.$$

The instantiated output state equals to the analysis result for the second context $(8, \mathtt{y})$ as we shown in Figure 17.

Localized Summary-based Analysis. We present the analysis algorithm for the localized summary-based analysis technique. The localized summary-based analysis technique consists of two steps: a generation step, and an instantiation step. Since a conditional summary of a function is generated by the analysis of the function, we use a top-down approach. We analyze a program from the beginning, and we generate a summary of a function on demand. The instantiation of the conditional summary effectively eliminates duplicated computation during the analysis.

Without loss of generality, we consider an abstract semantic function for a function $F^\sharp : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ that takes an input memory state and returns an output memory state of the function. We assume that a memory state is a map from variables to abstract values $\mathbb{M}^\sharp = \mathbb{X} \to_{\mathsf{fin}} \mathbb{V}^\sharp$. We write $i_A : A \to \mathbb{X}$ to denote the inclusion function of A in $\mathbb{X}$. The inclusion function of A is the restriction to A of the identity function on $\mathbb{X}$, which means $i_A = \mathrm{id}_{\mathbb{X}}|_A$ where $\mathrm{id}_{\mathbb{X}} : \mathbb{X} \to \mathbb{X}$ is an identity function. We write $F\backslash_a$ to denote $F \circ i_B$ where $B = \mathsf{Dom}(F) - \{\mathtt{x}\}$ $(\mathtt{x} \in \mathbb{X})$, and S^{c} for the complement of a set S.

Now, we define information that are written during the analysis of the abstract semantic function $F^\sharp$ with an input state $m^\sharp_{pre} \in \mathbb{M}^\sharp$:

Definition 7.1 (Defined Variables)**.** A variable $\mathtt{x} \in \mathbb{X}$ is defined during the analysis of the abstract semantic function $F^\sharp$ with an input memory state $m^\sharp_{pre}$ if the following property holds:

$$F^\sharp(m^\sharp_{pre})(\mathtt{x}) \neq m^\sharp_{pre}(\mathtt{x}).$$

A variable `x` is defined during the analysis of the semantic function $F^\sharp$ if and only if the value designated by the variable has been changed during the computation of the semantic function.

By the definition of defined variables, we can collect the set of defined variables $\mathcal{D} \in \mathcal{P}(\mathbb{X})$:

$$\mathcal{D} \stackrel{\mathsf{def}}{=} \{\mathtt{x} \in \mathbb{X} \mid F^\sharp(m^\sharp_{pre})(\mathtt{x}) \neq m^\sharp_{pre}(\mathtt{x})\}.$$

Example 7.1 (Defined Variables)**.** We consider the simple JavaScript program in Figure 16. During the analysis of function `k` with the following input state, variable rtn is defined variable because the analysis of `return h` statement at line 3 changes the value of variable rtn:

Prepared using speauth.cls
DOI: 10.1002/spe

$\{\texttt{h} \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"x"}, \texttt{t0} \mapsto 1\}$

3 **return** h;

$\{rtn^w \mapsto \texttt{"foo"}, \texttt{h} \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"x"}, \texttt{t0} \mapsto 1\}$

We define information that are read during the analysis of the semantic function $F^\sharp$ with an input state $m^\sharp_{pre} \in \mathbb{M}^\sharp$:

Definition 7.2 (Used Variables)**.** A variable $\texttt{x} \in \mathbb{X}$ is used during the analysis of the abstract semantic function $F^\sharp$ with an input memory state $m^\sharp_{pre}$ if the following property holds:

$$\forall \texttt{y} \in \mathcal{D},\ F^\sharp(m^\sharp_{pre}\backslash_{\texttt{x}})(\texttt{y}) \neq F^\sharp(m^\sharp_{pre})(\texttt{y}).$$

where $\mathcal{D}$ is the set of defined variables during the computation of $F^\sharp(m^\sharp_{pre})$.

A variable x is used during the analysis of the semantic function $F^\sharp$ if and only if the defined value in the result of the computation is not equal to the corresponding value in the result of the computation without the value of the variable x.

By the definition of used information, we can collect the set of used variables $\mathcal{U} \in \mathcal{P}(\mathbb{X})$:

$$\mathcal{U} \stackrel{\text{def}}{=} \{\texttt{x} \in \mathbb{X} \,|\, \forall \texttt{y} \in \mathcal{D},\ F^\sharp(m^\sharp_{pre}\backslash_{\texttt{x}})(\texttt{y}) \neq F^\sharp(m^\sharp_{pre})(\texttt{y})\}.$$

Example 7.2 (Used Variables)**.** We also consider the simple JavaScript program in Figure 16. During the analysis of function k with the following input state, variable h is used variable because the analysis of `return h` statement at line 3 requires the value of variable h and cannot produce the value of variable rtn without the value of variable h:

$\{\texttt{h} \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"x"}, \texttt{t0} \mapsto 1\}$

3 **return** h;

$\{rtn^w \mapsto \texttt{"foo"}, \texttt{h}^r \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"x"}, \texttt{t0} \mapsto 1\}$

We extend the semantic function to compute the def/use information also $F^\sharp_{du} : \mathbb{M}^\sharp \to \mathbb{M}^\sharp \times \mathcal{P}(\mathbb{X}) \times \mathcal{P}(\mathbb{X})$.

By the extended semantic function $F^\sharp_{du}$, we make a conditional summary of an input memory state $m^\sharp_{pre}$. The extended semantic function computes the original semantic function and its set of defined and used variables as follows:

$$F^\sharp_{du}(m^\sharp_{pre}) = (m^\sharp_{post}, \mathcal{D}, \mathcal{U})$$

where $F^\sharp(m^\sharp_{pre}) = m^\sharp_{post}$ and $\mathcal{D}$ and $\mathcal{U}$ are sets of defined and used variables for the input memory state $m^\sharp_{pre}$, respectively.

Definition 7.3 (Conditional Summary)**.** For an input memory state $m^\sharp_{pre}$, an extended semantic function $F^\sharp_{du}$, and the computation result of the function $F^\sharp_{du}(m^\sharp_{pre}) = (m^\sharp_{post}, \mathcal{D}, \mathcal{U})$, we define a conditional summary of the function as a following pair of a summary condition and a summary output:

$$(m^\sharp_{pre} \circ i_{\mathcal{U}}, m^\sharp_{post} \circ i_{\mathcal{D}}).$$

Example 7.3 (Conditional Summary Generation)**.** As an example, we consider the function k of the simple JavaScript program in Figure 16 and an input memory state $\{\texttt{h} \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"x"}, \texttt{t0} \mapsto 1\}$. As we have shown in Example 7.1 and 7.2, the defined and used variables for function k are $\{rtn\}$ and $\{\texttt{h}\}$, respectively. The output memory state is $\{rtn \mapsto \texttt{"foo"}, \texttt{h} \mapsto \texttt{"foo"}, \texttt{v} \mapsto \texttt{"x"}, \texttt{t0} \mapsto 1\}$. Then, the following pair is a conditional summary of the input memory state:

$$(\{\texttt{h} \mapsto \texttt{"foo"}\}, \{rtn \mapsto \texttt{"foo"}\}).$$

We collect all the summaries for function k in $\mathcal{S}^\sharp_{\mathtt{k}} : \mathbb{M}^\sharp \times \mathbb{M}^\sharp$ whenever we analyze the function.

When we have conditional summaries $\mathcal{S}^\sharp_{\mathtt{k}}$ and new input memory state $m^\sharp_{new}$ is given, we check a condition whether we can use the summary or not.

Definition 7.4 (Summary Instantiation Condition)**.** For a new input memory state $m^\sharp_{new} \in \mathbb{M}^\sharp$ and a summary $(m^\sharp_0, m^\sharp_1) \in \mathcal{S}^\sharp_{\mathtt{f}}$ of a function f, we can use the summary to compute the analysis result instead of computing the semantic function with the new input memory state $F^\sharp(m^\sharp_{new})$:

$$\forall \mathtt{x} \in \mathsf{Dom}(m^\sharp_0),\ m^\sharp_{new}(\mathtt{x}) = m^\sharp_0(\mathtt{x}).$$

Example 7.4 (Summary Instantiation)**.** As an example of the summary instantiation, we consider the input state for context $(8, \mathtt{y})$ with the summary we have created in Example 7.3. The given input state $\{\mathtt{h} \mapsto \texttt{"foo"}, \mathtt{v} \mapsto \texttt{"y"}, \mathtt{t0} \mapsto 2\}$ satisfies the summary instantiation condition because the value of variable h equals to the value in the summary and there is no other variable in the summary input state. Thus, we can generate the analysis result by overwriting the summary output $\{rtn \mapsto \texttt{"foo"}\}$ and the input state $\{\mathtt{v} \mapsto \texttt{"y"}, \mathtt{t0} \mapsto 2\}$. The generated output state $\{rtn \mapsto \texttt{"foo"}, \mathtt{v} \mapsto \texttt{"y"}, \mathtt{t0} \mapsto 2\}$ is equal to the analysis result as we described in Figure 17.

Given input state satisfies the summary instantiation condition for the summary $(m^\sharp_0, m^\sharp_1) \in \mathcal{S}^\sharp_{\mathtt{f}}$, we can compute the output memory state of the function f by instantiating the summary as follows:

$$m^\sharp_1 \sqcup m^\sharp_{new} \circ i_{\mathcal{D}^c}.$$

Theorem 7.1 (Correctness)**.** For an abstract semantic function $F^\sharp : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ of a function f and a summary $(m^\sharp_0, m^\sharp_1) \in \mathcal{S}^\sharp_{\mathtt{f}}$, the following property holds:

$$\forall m^\sharp \in \mathbb{M}^\sharp,\ \forall \mathtt{x} \in \mathsf{Dom}(m^\sharp_0),\ m^\sharp_0(\mathtt{x}) = m^\sharp(\mathtt{x}) \Rightarrow F^\sharp(m^\sharp) = m^\sharp_1 \sqcup m^\sharp \circ i_{\mathcal{D}^c}$$

where $\mathcal{D} = \mathsf{Dom}(m^\sharp_1)$.

When we have an input state $m^\sharp$ for an abstract semantic function $F^\sharp$ of a function f, we have a conditional summary that can be instantiated, we instantiate the summary to compute the analysis result of the function f to avoid the duplicated computations of the function. Otherwise, we analyze the function and make new summary for the function. By the theorem 7.1, we guarantee that the analysis result that we can obtain from the instantiation of a summary is equal to the analysis result by new input memory state $m^\sharp_{new}$.

8. EVALUATION

This section seeks for an experimental validation that the composite abstraction is necessary and useful for the computation of precise call-graph information. We first introduce our composite abstraction based JavaScript static analyzer CompAbs. Then, we perform two sets of experiments: first, we compare the composite abstraction with state-of-the-art analyzers TAJS and $\mathsf{SAFE}_{\mathsf{LSA}}$ using micro-benchmarks that represent different categories of FCT patterns; second, we assess the contribution of each layer of our composite abstraction, and of the localized summary-based analysis technique, using jQuery benchmarks as an evaluation subject.

8.1. Experimentation settings

We implemented the analyzer CompAbs, based on our new composite abstraction. CompAbs was built on top of the open-source JavaScript analysis framework SAFE [2].

CompAbs supports several parameters to enable precise abstractions, that we describe in the following paragraphs.

CompAbs is fully context sensitive except for recursive calls, which are unrolled at most j times, where j is a parameter of the analysis. Dynamic unrolling of loops is also bounded by a parameter k.

DOI: 10.1002/spe

Loop	FCT Pattern	TAJS	SAFE$_{LSA}$	CompAbs
`while`	Singular	✓	✓	✓
`for-in`	Singular	✓	✓	✓
`while`	Plural	✗	✗	✓
`for-in`	Plural	✗	✗	✓

Table I. The result of the micro benchmarks shows whether each analyzer can precisely infer the field corresepondence relation or not. The experiment used `while` and `for-in` loops that contains either a singular or a plural FCT pattern.

The number of nested partitions for nested FCT patterns is bounded by a parameter l. Indeed, the partitioning abstraction as we formalized it in Section 5.1 handles a single FCT patterns, so that it would not suffice in the case of nested FCT patterns. When several nested patterns are nested, CompAbs needs to utilize imbricated partitions, which may become too costly. Thus, we let the analysis distinguish at most the l last partitions created, which is adequate to handle up to l nested FCT patterns.

We used different parameters for each experiment. For the first set of experiments, we choose the simpliest parameters that only shows the benefit from our composite abstraction. For the second set of experiments, we choose the parameters by experiments to obtain the precise pre-condition that can reduce the analysis of a plural FCT patterns to that of a singular one. The second set of experiments with the parameter shows the necessity of our approach.

To describe strings, CompAbs utilizes an abstraction based on creation sites, and on a reduced product of the domain of string constants and of a domain that abstract strings using prefixes.

CompAbs supports its own DOM modeling. Unlike the existing DOM models in SAFE, we precisely support specific DOM modeling for Chrome browser version 55 and event models for initialization such as `readystatechange`, `DOMContentLoaded`, and `load` events. Thus, the analysis result will not compute an over-approximation of features of an arbitrary set of browsers, but will compute a precise over-approximation of the executions observed with one specific browser. We tested our DOM modeling during the experiment on jQuery programs, and faithfully implemented DOM models to compute over-approximation of the real behaviors of the browser. We represent the initial DOM tree as corresponding abstract DOM tree, which precisely keep all the information of the initial DOM tree. On the other hand, we abstract a set of dynamically generated DOM elements to an abstract DOM element depending on its allocation site.

We performed all the experiments on a Linux machine with 4.0GHz Intel Core i7-6700K CPU and 32GB of RAM.

8.2. Analysis of micro benchmarks with different loop patterns

The goal of this experiment is to observe the output of the current versions of the state-of-the-art JavaScript static analyzers TAJS and SAFE$_{LSA}$ for each category of FCT patterns introduced in Section 2.

We ran TAJS, SAFE$_{LSA}$, and CompAbs on a number of small diagnostic benchmarks, each of which consists of a loop containing an FCT pattern. We manually wrote the test cases to assess whether an analysis can precisely infer a field correspondence relation or not for each kinds of loops containing an FCT pattern. We used four micro benchmarks, with all combinations of loop types (`while` and `for-in`), and of FCT patterns (singular and plural). These four benchmarks do not use DOM-related features. They are set up so that each FCT pattern is being classified in the same category in all three analyzers TAJS, SAFE$_{LSA}$, and CompAbs. At the end of each loop, we attempt to verify the precise field correspondence relation specified by the equality between fields `o.x` $\neq$ `o.y`. To make sure that the analysis results can be compared in a fair manner, CompAbs used $j = 0$, $k = 0$, and $l = 1$ as parameters, which leads to merging all recursive calls, using no dynamic unrolling, and distinguishing at most one FCT pattern.

Table I summarizes the results produced by the analysis of these four micro benchmarks. It shows whether each analyzer can precisely infer the field correspondence relation or not. The first and

DOI: 10.1002/spe

second columns show the loop kind and FCT pattern category of each program. The remaining columns report whether each analysis successfully infers the precise field correspondence relation or not. TAJS and SAFE_{LSA} fail to infer precise field correspondence relations in the case of plural FCT patterns, which is consistent with our observation that such loops cannot be addressed by using only the unrolling techniques implemented in these tools.

This confirms that the existing analyzers fail to infer precise field correspondence relations for loops with an FCT pattern, which us classified as a plural.

Although our composite abstraction can also precisely analyze a plural FCT pattern that does not occur inside a loop, we could not find such cases in practice, thus we did not include such a test case in our experiment.

8.3. Analysis of jQuery tutorials

The goal of this experiment is to evaluate the necessity and usefulness of each of the three layers of the composite abstraction and of the optimization technique based on conditioned summaries, (Section 7) using jQuery tutorial benchmarks. We compare the call-graph information of each analysis.

We used the abstractions supported by SAFE as a baseline abstraction for this experiment. The baseline analysis does not use partitioning abstraction, and uses a basic object abstraction without disjunctions (unlike the object abstraction shown in Section 5.2). The baseline analysis also does not use creation-site based string abstraction. Instead, it uses only the reduced product of a constant string abstraction and a prefix string abstraction.

We measure the number of programs that have plural FCT patterns so as to assess the necessity of our approach. We also measure the number of programs that can be analyzed in a given time limit so as to assess the scalability. We measure the number of spurious call-sites and spurious call-edges in the result so as to assess the precision. Last, we measure the percentage of coverage so as to assess the coverage of the analysis result. Note that our analysis does not support the `eval` function that constructs code at runtime and executes it dynamically. Thus, the analysis does not cover the executions caused by the `eval` function by considering the function call as a no operation.

To evaluate the analysis of FCT patterns in framework libraries, we use the jQuery benchmarks from the experiments performed with TAJS in [5]. This work relies on 71 programs chosen amont the jQuery tutorial[†], that perform simple operations using jQuery 1.10.0. In order to minimize the impact of DOM modeling on the analysis results, our experiment uses jQuery 1.4.4 that supports all the features required for the benchmarks while using fewer DOM-related features than jQuery 1.10.0. We have also made sure that our DOM model supports all the DOM-related features used by the program in our benchmark suite. The programs are about 7430 lines of code including jQuery.

The results are presented in Table II.

Need for the composite abstraction. First, to validate the need for the abstractions that we propose, we measure how many programs have plural FCT patterns. CompAbs uses $j = 2$, $k = 30$, and $l = 3$ as parameters, which leads to unrolling recursive calls twice, bounding dynamic unrolling depth to 30 (when it can be applied), and distinguishing at most the last 3 sets of partitions that can be caused by 3-nested FCT patterns. We chose the parameters by experiments to compute a precise pre-condition that can reduce the analysis of a plural FCT pattern to that of a singular one. Among 71 benchmarks, we observe that 27 programs have at least one plural FCT pattern. Thus, about 38% of the benchmarks need to handle plural FCT patterns, which may require our composite abstraction (as shown in Section 8.2).

Precision. Second, to assess the analysis precision, we measure the number of spurious call edges in analysis results. We identify spurious call edges in the analysis result of a program by comparison with a dynamic call graph constructed during the concrete execution of the instrumented program in the Chrome web browser. Since the benchmarks are simple and we ran them several times trying to

[†] http://www.jquery-tutorial.net/

DOI: 10.1002/spe

Table II. Scalability and precision for call graph computations compared to each combination of three layers of the composite abstraction. The first column shows the setting of each analysis: P denotes the partitioning abstraction, O denotes the object abstraction, S denotes the creation-site based string abstraction, and L denotes the localized summary-based analysis technique. For example, P+S denotes the extended baseline analysis with our partitioning abstraction and creation-site based string abstraction. The second column shows the number of programs that finished analysis in less than 8 hours, which effectively measures the scalability of the analysis. Note that all the abstract domain combinations that are not presented in this table do not complete analysis of *any* of these 71 programs within 8 hours. The other columns measure the analysis precision; SE denotes the number of spurious call edges on average, SC denotes the number of spurious call sites on average, and Cov denotes the coverage ratio of actual call sites.

	Success	SE	SC	Cov(%)
P+S	36 / 71	3.15	39.7	97.21
P+O	68 / 71	0.03	16.4	99.76
P+O+S	69 / 71	0.03	15.3	99.97
P+O+S+L	71 / 71	0.03	14.9	99.97

cover all the possible execution flows, we assume that the dynamic call graphs contain a complete set of call edges. Then, we count the number of call sites for which there exists more than one false edge, i.e., of a call edge reported by the analysis but that occurs in no concrete execution in our sample. Note that the call sites for which there exists one false edge are counted as spurious call sites and they are not counted as spurious edges.

We found spurious call edges in eight subjects (39, 40, 44, 47, 50, 53, 69, and 71) out of 71. We manually investigated them, and observed that dynamic call graphs for five subjects (40, 44, 50, 53, 71) were incomplete because of an HTML parsing problem in the Chrome browser. We confirmed that the spurious call edges from the five subjects disappeared after we revised the HTML documents to bypass the parsing problem. We found that each of the remaining 3 subjects had one callsite with spurious call edges, all of which are due to imprecise DOM modeling or infeasible execution flows. Because our DOM modeling does not consider the precise semantics of browser layout engines, the analysis cannot precisely analyze several DOM-related functions such as `querySelectorAll` or `getComputedStyle`. Imprecise analysis results of such function calls affect the analysis results of execution flows, which may cause spurious call edges.

The third to the fifth columns of Table II summarize the precision measurements of 71 benchmarks; SE denotes the number of spurious call edges on average, and SC denotes the number of spurious call sites on average; Cov denotes the coverage ratio of actual call sites. As our analyzer does not support the `eval` function that is extremely hard to account for precisely, it may fail to cover all executions of a given program even though it is designed so as to handle all other JavaScript features in a sound manner. Therefore, the analysis of a given program may fail to achieve 100% of coverage. Moreover, the results of analysis runs that fail to complete within 8 hours do not cover all executions of the program. In this case, the actual number of spurious call edges may be larger than the one in the table.

All the combinations of abstract domains that are not shown in Table II do not complete on any of the 71 programs within the 8 hours timeout. The main issue that hinders the analysis is the large number of spurious callees. The baseline analysis that does not use our composite abstraction gave up the analysis of 50 out of 71 programs due to a large number of spurious call edges. Our analyzer gives up to analyze a program when there are more than 200 function calls at one user call-site. For the rest of the 21 programs, the baseline analysis could cover 74.15% of actual call sites in average within 8 hours due to the computation of a large number of spurious call edges.

Soundness. Third, we checked the soundness of CompAbs using dynamic call graphs. The dynamic call graphs include all the built-in function calls and callbacks during the execution of built-in and DOM related functions. However, we excluded callbacks for built-in functions during the execution of the following built-in functions: `RegExp.prototype.test`, `RegExp.prototype.replace`, and `RegExp.prototype.split`. They internally call other built-in functions. For example, `RegExp.prototype.test` internally calls

`RegExp.prototype.exec`. These call relations are strongly related to the browser implementation, but are not related to user program behaviors.

Our study confirms that static call graphs of 70 programs out of 71 subsume the corresponding dynamic call graphs. Because the subject (4) uses `eval` that constructs code at runtime and executes it dynamically, static call graphs could not cover dynamically constructed call edges, for which CompAbs reports possible coverage warnings in such cases.

Scalability. Forth, to compare the analysis scalability, we measure the number of programs the analysis of which terminates in less than 8 hours by each combination of three layers of our composite abstraction and the optimization technique in CompAbs. The second column in Table II summarizes the scalability measurements; out of 71 benchmarks, the analysis without our object abstraction completes on 36 programs among 71 within 8 hours; the analysis without our creation-site based string abstraction completes on 68 programs among 71; the analysis with all of the composite abstraction completes on 69 programs; and finally, the analysis with the full composite abstraction and the optimization based on conditioned summaries completes on all 71 programs in our set of experiments. All the combinations of abstract domains that are not shown in the table do not complete on any of the 71 programs within the 8 hours timeout. These results confirm the three components of the composite abstraction are all necessary to ensure that the analysis scales up. The conditioned summary also improves the overall analysis scalability and precision that we describe in the following separate paragraphs.

Effectiveness of conditioned summaries. Last, we assess the usefulness of our localized summary-based analysis technique. To show the usefulness of our optimization technique, we compare the analysis time of the composite abstraction with/without the optimization technique. CompAbs without the conditioned summaries completes the analysis of each of the 64 programs within 6 minutes. Over these program the average analysis time is 27 seconds (minimum, median, and maximum are 5, 9, and 315 seconds, respectively). Among the 7 other programs, CompAbs analyzes each of 4 programs within 8 hours, and analyzed each of the rest of 3 programs within 2.1 days. On the other hand, CompAbs with conditionned summaries enabled completes the analysis of each of these 64 programs within 5 minutes, and the average time drops down to 19 seconds (minimum, median, and maximum are 4, 6, and 241 seconds, respectively). Among the 7 other programs, CompAbs analyzes each of 4 programs within 2 hours, and analyzes each of the remaining 3 programs within 8 hours.

The analysis optimization technique based on conditioned summaries reduces the analysis time by 30% on average. In the best case, the analysis time is reduced by 96%. In this case, the analysis takes 38.8 hours without conditioned summaries and only 1.6 hours when conditioned summaries are enabled. We also observed a single case of slow down: for one program among the 71 examples, the analysis time is increased by 94% due to the overhead inherent in the conditioned summaries. In all the other cases, the use of conditioned summaries yield to a faster analysis. The subject P+O+S+L shows that the precision of the optimized analysis was slightly improved compared to the subject P+O+S in Table II. The precision improvement came from the benefit of the localization technique. The localization technique avoids the propagation of the output state which are not relevant to the function body through spurious cycles at the end of a function, and it improves the precision [16, 19].

9. RELATED WORK

SAFE [2] is an analysis framework for JavaScript web applications, which supports DOM modeling [20] and web API misuse detection [21], among others. It is based on abstract interpretation, and supports various analysis sensitivities including k-CFA, parameter sensitivity, object sensitivity, and loop-sensitive analysis (LSA) [6]. While its aggressively unrolling LSA works well for programs using simple loops with bounded iterations, we showed that it does not scale for complex loops with unbounded numbers of iterations or imprecise pre-conditions.

TAJS [1, 5] is an open-source static analyzer for JavaScript, which supports DOM modeling [22]. It uses a simple object abstraction, which can be represented by a limited form of conjunctions, without disjunctions. While the object abstraction is clearly designed and highly tuned for their own string abstraction, it does not support various string abstractions for a field name abstraction. In addition, as we described in Section 3, it may lose precision when it disambiguates field correspondence relations after joining the states from loop iterations.

HOO [23] computes a very different abstraction of JavaScript objects, which captures abstract forms of field relations. Indeed, HOO lets symbolic set variables denote unbounded size sets of fields of JavaScript objects to express logical facts on these sets, using specific set abstract domains [24]. For instance, it can express that two objects have exactly the same set of fields, or that the fields of a first object are included into the fields of a second object. Such program invariants are useful to reason over libraries that may be used in a general and unknown context, or in order to implement a modular verifier. By contrast, this approach is less adequate to compute precise information about the result of long and complex initialization routines like the codes our analysis targets.

WALA [25] supports static analysis of JavaScript programs. It performs a conventional propagation-based pointer analysis with techniques that can handle some of the FCT patterns [7, 26].

ACG [27] is a field-based, intentionally unsound call graph analysis. The analysis ignores dynamic field lookup and update instructions, does not track non-functional values, and does not distinguish JavaScript objects since it abstracts all concrete objects into a single abstract object. Thus, ACG achieves a practical level of scalability and precision. Since ACG does not compute an over-approximation of program behaviors, the analysis is not appropriate to verify safety properties in a program. On the other hand, it is useful to support development tools.

10. CONCLUSION

We presented a composite abstraction that can capture precise field correspondence relations by local reasoning about FCT patterns. The composite abstraction consists of three layers of abstractions: the partitioning abstraction that supports separate reasoning about the executions in an FCT pattern, the object abstraction describes accurate properties over fields, so as to disambiguate the field correspondence relations even after abstract joins, and the string abstraction is precise enough to disambiguate field names in an object. While most existing JavaScript analyses unroll loops aggressively, which does not scale for loops that cannot be unrolled, our analysis can infer field correspondence relations precisely even for loops which cannot be unrolled. The results of experiments based on a prototype implementation show that the analysis can avoid a large number of spurious callees caused by the precision loss of the field correspondence relation in jQuery examples.

REFERENCES

1. Jensen SH, Møller A, Thiemann P. Type analysis for JavaScript. *SAS*, 2009.
2. Lee H, Won S, Jin J, Cho J, Ryu S. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. *FOOL*, 2012.
3. Hackett B, Guo Sy. Fast and precise hybrid type inference for JavaScript. *PLDI*, New York, NY, USA, 2012.
4. Wei S, Ryder BG. Practical blended taint analysis for JavaScript. *ISSTA*, 2013.
5. Andreasen E, Møller A. Determinacy in static analysis for jQuery. *OOPLSA*, 2014.
6. Park C, Ryu S. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. *ECOOP*, 2015.
7. Sridharan M, Dolby J, Chandra S, Schäfer M, Tip F. Correlation tracking for points-to analysis of JavaScript. *ECOOP*, 2012.
8. Wei S, Tripp O, Ryder BG, Dolby J. Revamping javascript static analysis via localization and remediation of root causes of imprecision. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, ACM: New York, NY, USA, 2016; 487–498, doi:10.1145/2950290.2950338. URL http://doi.acm.org/10.1145/2950290.2950338.
9. Kashyap V, Dewey K, Kuefner EA, Wagner J, Gibbons K, Sarracino J, Wiedermann B, Hardekopf B. JSAI: A static analysis platform for JavaScript. *FSE '14: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.
10. Balakrishnan G, Reps T. Recency-abstraction for heap-allocated storage. *SAS*, 2006.
11. Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL*, 1977.

12. European Association for Standardizing Information and Communication Systems (ECMA). ECMA-262: ECMAScript Language Specification. Edition 5.1 2011.
13. Cousot P, Cousot R. Systematic design of program analysis frameworks. *POPL*, 1979.
14. Jones ND, Muchnick SS. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, ACM: New York, NY, USA, 1982; 66–74, doi:10.1145/582153.582161. URL http://doi.acm.org/10.1145/582153.582161.
15. Chase DR, Wegman M, Zadeck FK. Analysis of pointers and structures. *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, ACM: New York, NY, USA, 1990; 296–310, doi:10.1145/93542.93585. URL http://doi.acm.org/10.1145/93542.93585.
16. Oh H, Brutschy L, Yi K. *Access Analysis-Based Tight Localization of Abstract Memories*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2011; 356–370, doi:10.1007/978-3-642-18275-4_25. URL https://doi.org/10.1007/978-3-642-18275-4_25.
17. Ko Y, Heo K, Oh H. A sparse evaluation technique for detailed semantic analyses. *Computer Languages, Systems & Structures* 2014; .
18. Oh H, Heo K, Lee W, Lee W, Yi K. Design and implementation of sparse global analyses for C-like languages. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012; 229–238.
19. Oh H. Large spurious cycle in global static analyses and its algorithmic mitigation. *APLAS*, 2009.
20. Park C, Won S, Jin J, Ryu S. Static analysis of JavaScript web applications in the wild via practical DOM modeling. *ASE*, 2015.
21. Bae S, Cho H, Lim I, Ryu S. $\mathsf{SAFE}_{\mathsf{WAPI}}$: Web API misuse detector for web applications. *ESEC/FSE*, 2014.
22. Jensen SH, Madsen M, Møller A. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. *ESEC/FSE*, 2011.
23. Cox A, Chang BYE, Rival X. Automatic analysis of open objects in dynamic language programs. *SAS*, 2014.
24. Cox A, Chang BYE, Sankaranarayanan S. QUIC graphs: Relational invariant generation for containers. *ECOOP*, 2013, doi:10.1007/978-3-642-39038-8_17.
25. IBM Research. T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net.
26. Schäfer M, Sridharan M, Dolby J, Tip F. Dynamic determinacy analysis. *PLDI*, 2013.
27. Feldthaus A, Schäfer M, Sridharan M, Dolby J, Tip F. Efficient construction of approximate call graphs for JavaScript IDE services. *ICSE*, 2013.